



# Implementing a Physical Units Library for C++

Mateusz Pusz  
April 12, 2019

?

!

Why?

Why  
Not?

Famous motivating example

?

!

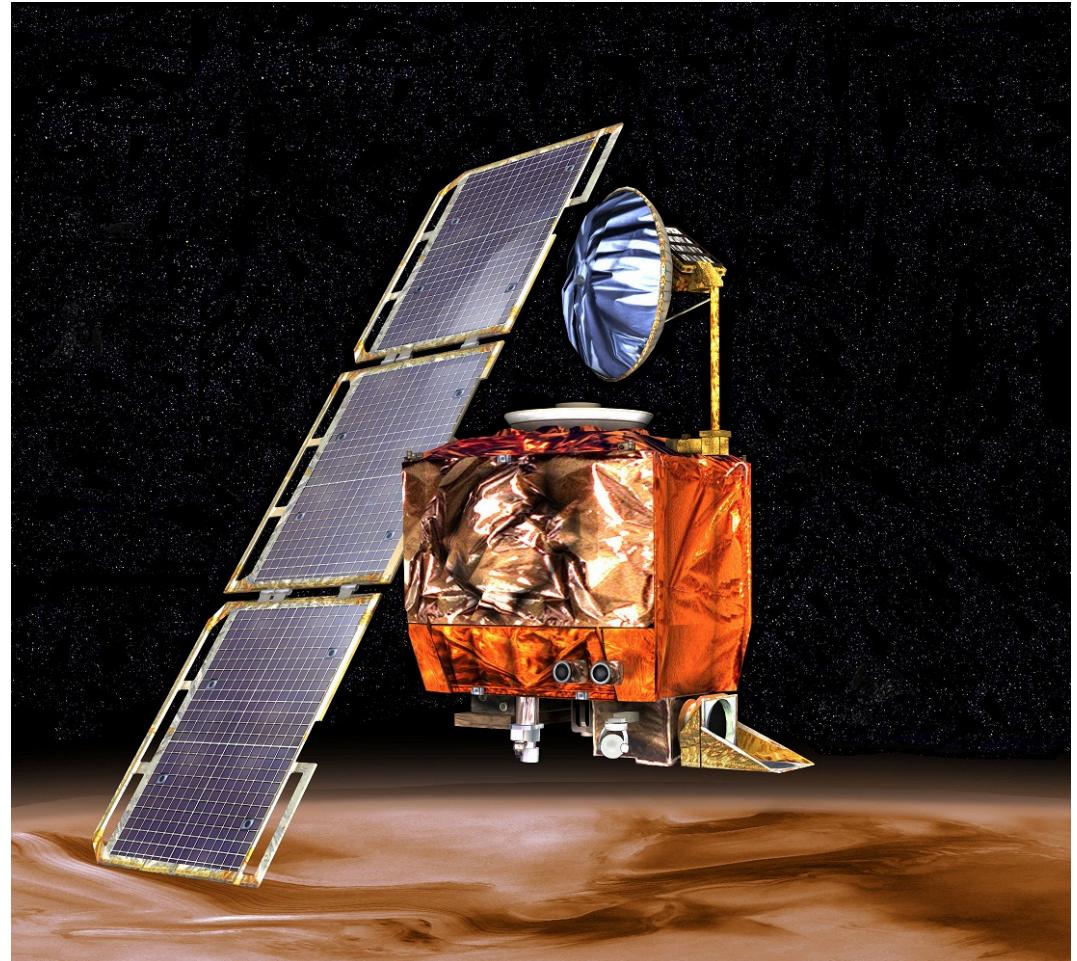
Why?

Why  
Not?

# The Mars Climate Orbiter

---

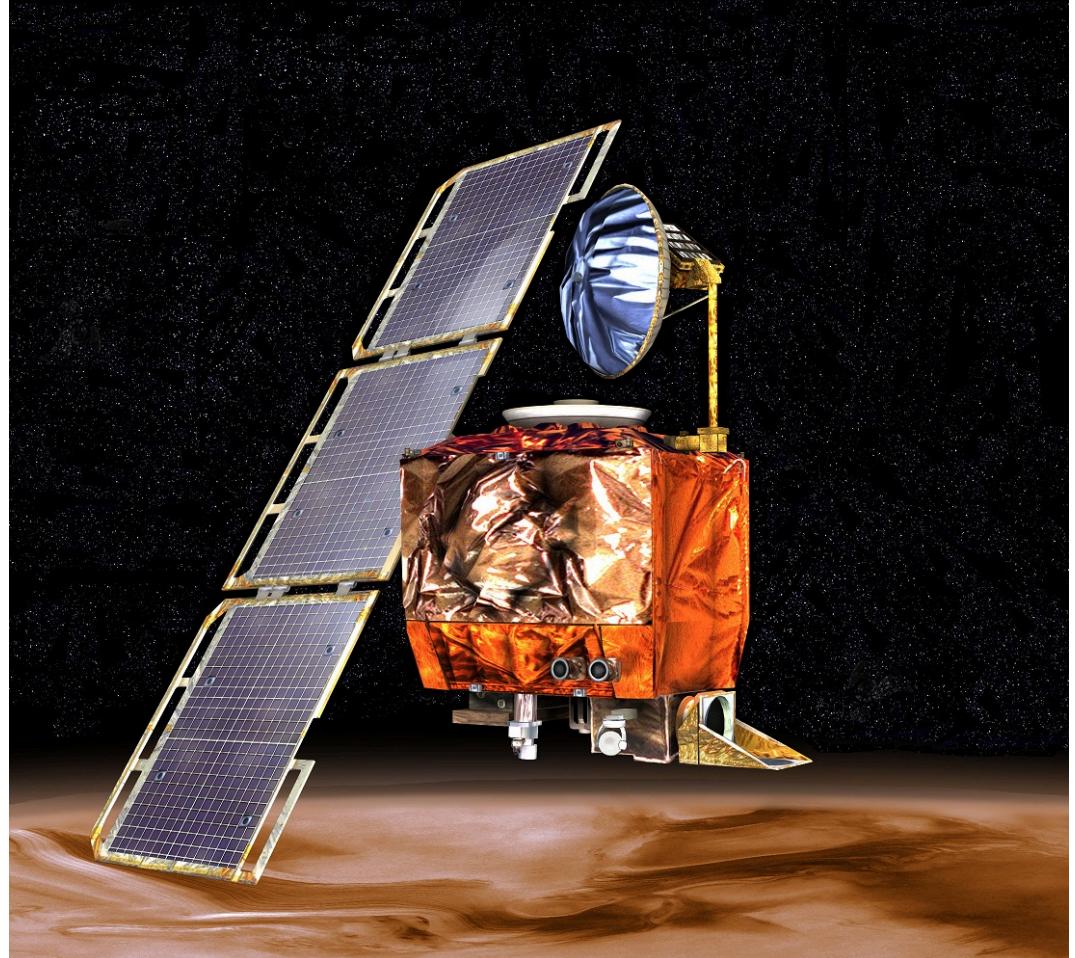
- Robotic space probe launched by NASA on December 11, 1998



# The Mars Climate Orbiter

---

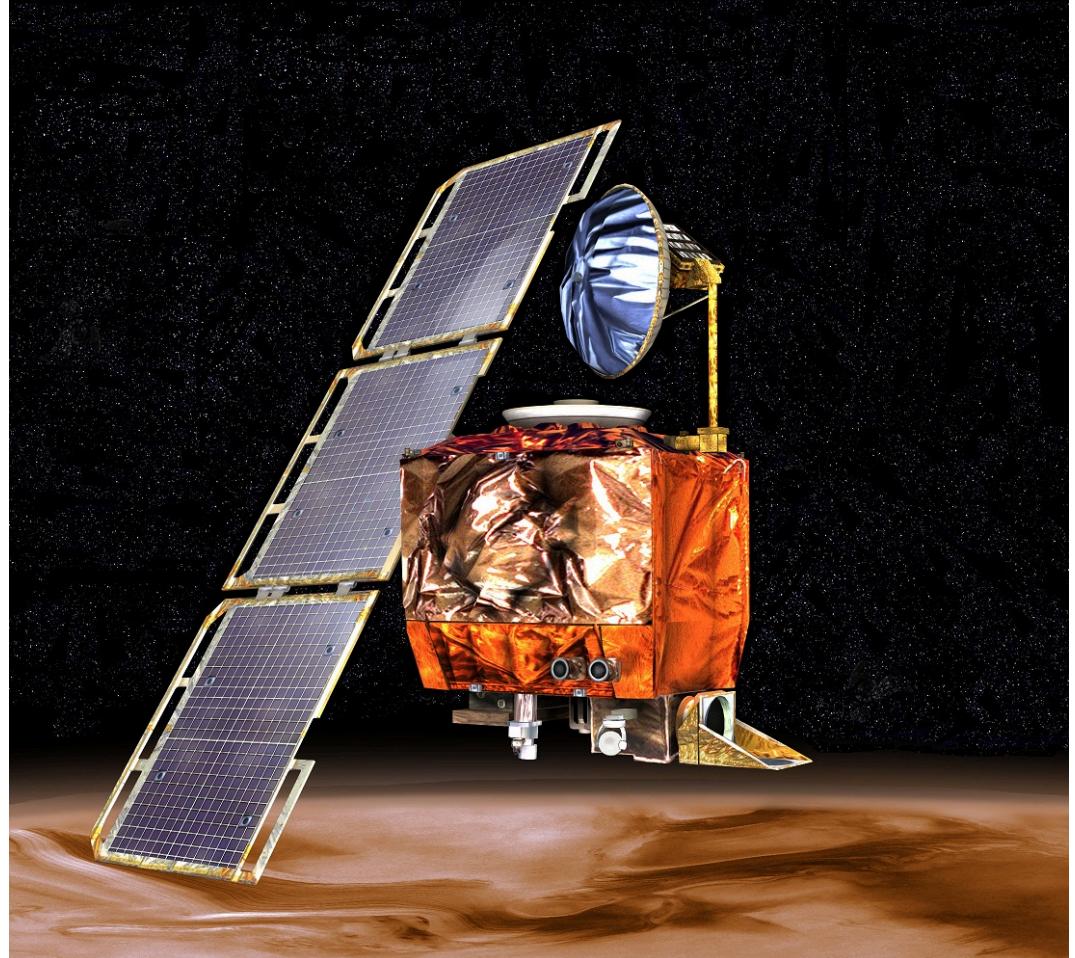
- Robotic space probe launched by NASA on December 11, 1998
- Project costs: **\$327.6 million**
  - spacecraft development: \$193.1 million
  - launching it: \$91.7 million
  - mission operations: \$42.8 million

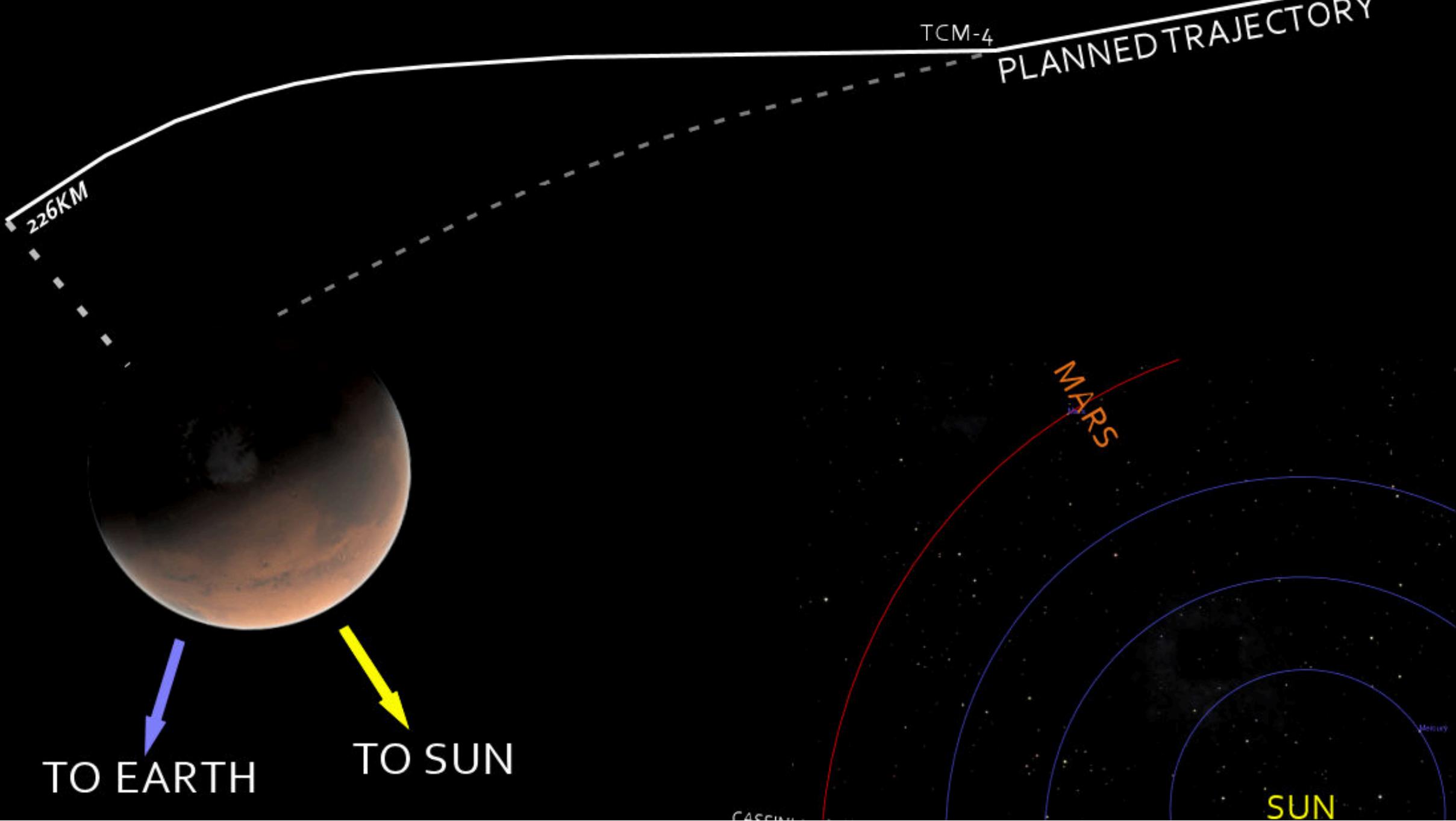


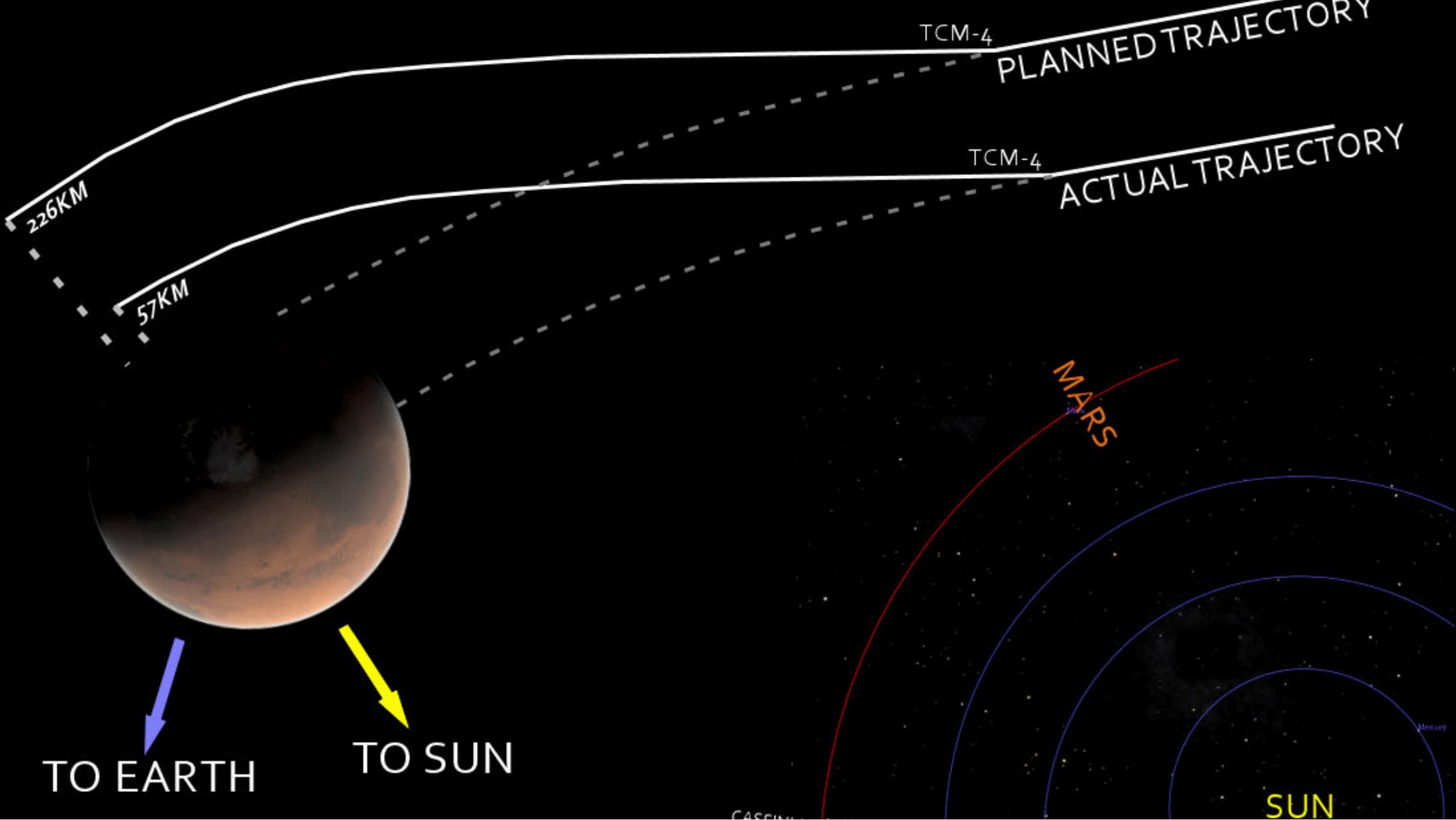
# The Mars Climate Orbiter

---

- Robotic space probe launched by NASA on December 11, 1998
- Project costs: **\$327.6 million**
  - spacecraft development: \$193.1 million
  - launching it: \$91.7 million
  - mission operations: \$42.8 million
- Mars Climate Orbiter began the planned *orbital insertion maneuver* on September 23, 1999 at 09:00:46 UTC







# The Mars Climate Orbiter

---

- Space probe went **out of radio contact** when it passed behind Mars at 09:04:52 UTC, *49 seconds* earlier than expected
- Communication was never reestablished
- The **spacecraft disintegrated** due to atmospheric stresses

# What went wrong?

---

# What went wrong?

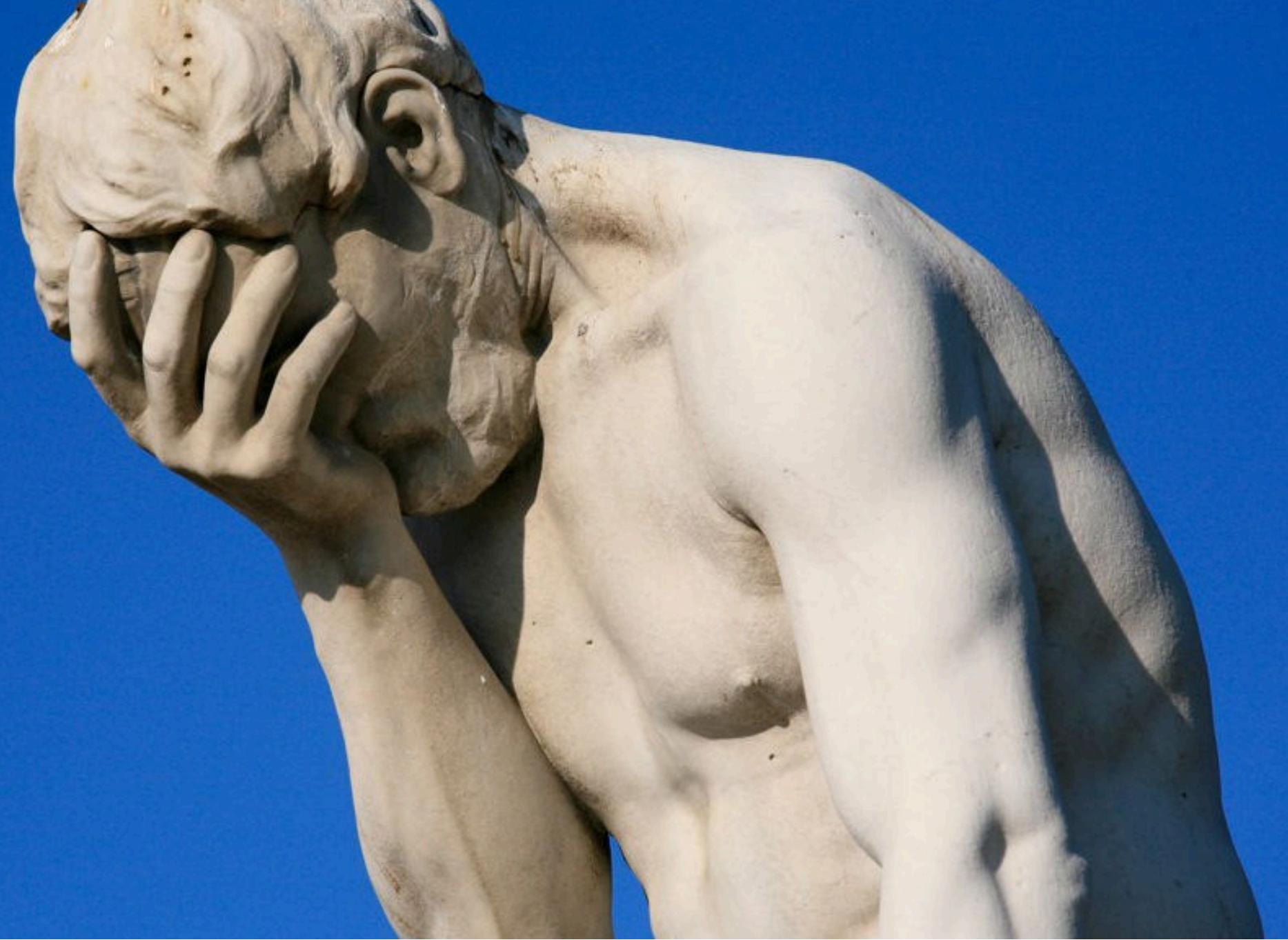
---

- The **primary cause** of this discrepancy was that
  - one piece of ground software supplied by Lockheed Martin produced results in a *United States customary unit, contrary to its Software Interface Specification* (SIS)
  - second system, supplied by NASA, expected those results to be in *SI units, in accordance* with the SIS

# What went wrong?

---

- The **primary cause** of this discrepancy was that
  - one piece of ground software supplied by Lockheed Martin produced results in a *United States customary unit, contrary to its Software Interface Specification* (SIS)
  - second system, supplied by NASA, expected those results to be in *SI units, in accordance* with the SIS
- Specifically
  - software that calculated the total impulse produced by thruster firings calculated results in **pound-seconds**
  - the trajectory calculation software then used these results to update the predicted position of the spacecraft and expected it to be in **newton-seconds**



# Human error occurs all the time

---

Human error occurs all the time. But even so *we have a tremendous success rate because we have systems that detect and correct the errors.* The problem here is that our system failed to do that.

-- *Carl Pilcher*  
*science director at NASA*

?

!

ANOTHER EXAMPLE

Why?

Why  
Not?

Why do I care?

?

!

Why?

Why  
Not?

# A long time ago in a galaxy far far away...

---

# A long time ago in a galaxy far far away...

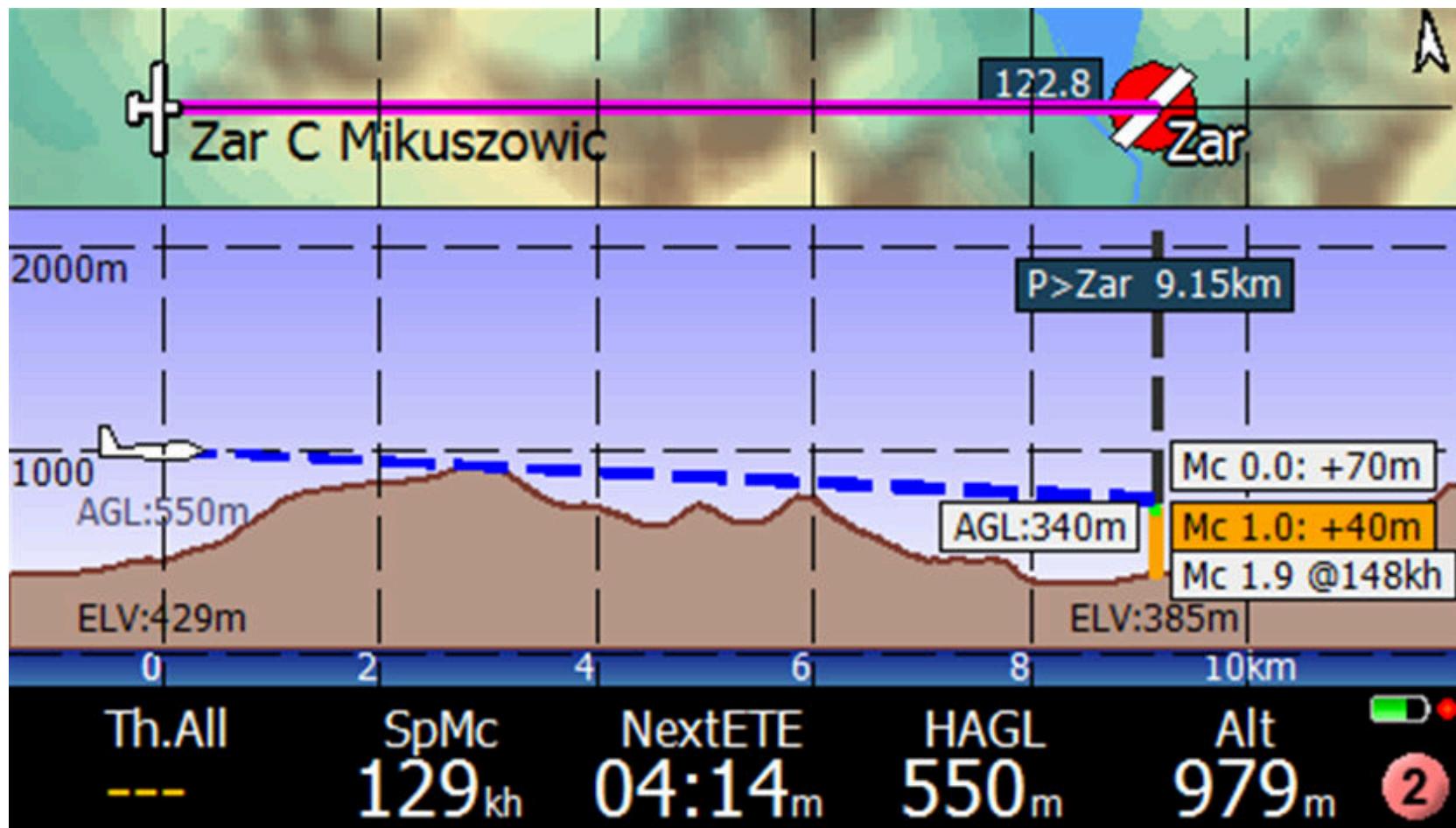
---



# Tactical Flight Computer



# Tactical Flight Computer



# What is the correct order?

---

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

# What is the correct order?

---

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

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

# What is the correct order?

---

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

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

# double - an ultimate type to express quantity

```
double GlidePolar::MacCreadyAltitude(double emcready,  
                                     double Distance,  
                                     const double Bearing,  
                                     const double WindSpeed,  
                                     const double WindBearing,  
                                     double *BestCruiseTrack,  
                                     double *VMacCready,  
                                     const bool isFinalGlide,  
                                     double *TimeToGo,  
                                     const double AltitudeAboveTarget,  
                                     const double cruise_efficiency,  
                                     const double TaskAltDiff);
```

# We shouldn't write the code like that anymore

---

```
// Air Density(kg/m3) from relative humidity(%),
// temperature(°C) and absolute pressure(Pa)
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)));
}
```

**DID YOU EVER HAVE TO WRITE THE CODE THIS WAY?**

# Why do we write our code this way?

---

# Why do we write our code this way?

---

- No support in the C++ Standard Library
  - `std::chrono` helped a lot for time and duration
  - `date` support comes in C++20
  - still not enough for full dimensional analysis

# Why do we write our code this way?

---

- No support in the C++ Standard Library
  - `std::chrono` helped a lot for time and duration
  - `date` support comes in C++20
  - still not enough for full dimensional analysis
- Lack of good alternatives
  - poor user experience (i.e. compilation errors)
  - heavy dependencies (i.e. Boost.Units)
  - custom 3rd party libraries often not allowed in production code

# Why do we write our code this way?

---

- No support in the C++ Standard Library
  - `std::chrono` helped a lot for time and duration
  - `date` support comes in C++20
  - still not enough for full dimensional analysis
- Lack of good alternatives
  - poor user experience (i.e. compilation errors)
  - heavy dependencies (i.e. Boost.Units)
  - custom 3rd party libraries often not allowed in production code
- Implementing a good library by ourselves is hard

# Why do we write our code this way?

---

- No support in the C++ Standard Library
  - `std::chrono` helped a lot for time and duration
  - `date` support comes in C++20
  - still not enough for full dimensional analysis
- Lack of good alternatives
  - poor user experience (i.e. compilation errors)
  - heavy dependencies (i.e. Boost.Units)
  - custom 3rd party libraries often not allowed in production code
- Implementing a good library by ourselves is hard

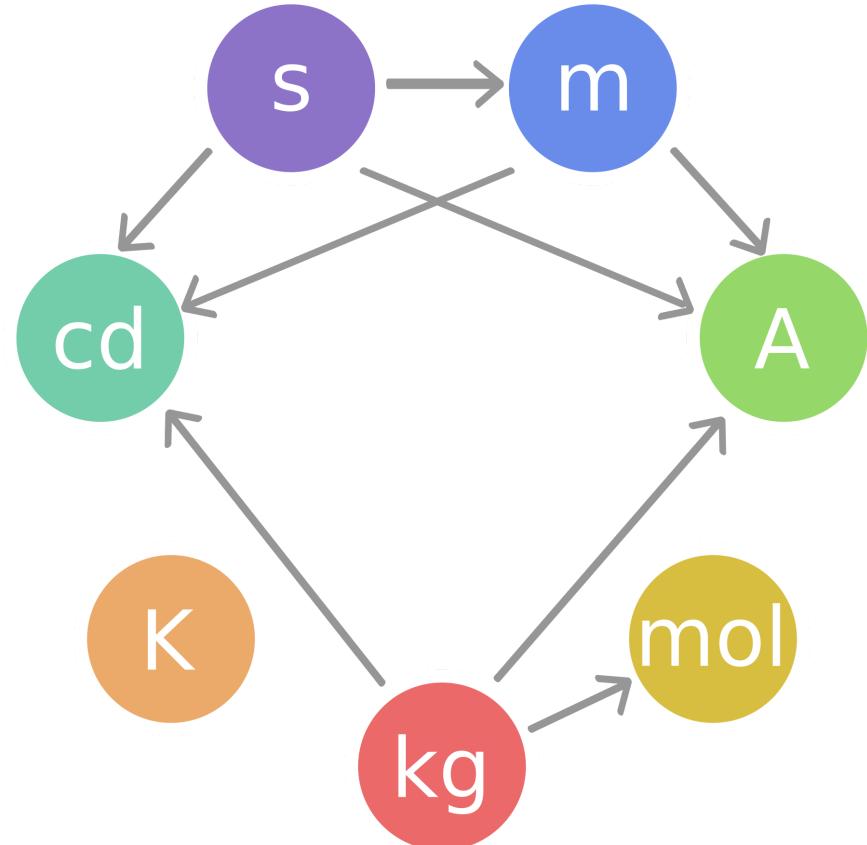
Let's do something about that!

## TERMS AND DEFINITIONS

# International System of Units (SI)

---

- 7 base units
- 22 named units
- 20 prefixes to the unit names and unit symbols



# SI Base Units

---

QUANTITY	DIMENSION SYMBOL	UNIT SYMBOL	UNIT NAME
time	T	s	second
length	L	m	metre
mass	M	kg	kilogram
electric current	I	A	ampere
thermodynamic temperature	Θ	K	kelvin
amount of substance	N	mol	mole
luminous intensity	J	cd	candela

# Examples of SI derived units expressed in SI Based Units

QUANTITY	DIMENSION SYMBOL	UNIT SYMBOL	UNIT NAME
area	A	$\text{m}^2$	square metre
volume	V	$\text{m}^3$	cubic metre
velocity	v	$\text{m}\cdot\text{s}^{-1}$	metre per second
acceleration	a	$\text{m}\cdot\text{s}^{-2}$	metre per second squared
density	$\rho$	$\text{kg}\cdot\text{m}^{-3}$	kilogram per cubic metre
magnetic field strength	H	$\text{A}\cdot\text{m}^{-1}$	ampere per metre
luminance	Lv	$\text{cd}\cdot\text{m}^{-2}$	candela per square metre

# Examples of SI derived units with special name

QUANTITY	UNIT SYMBOL	UNIT NAME	IN OTHER SI UNITS	IN SI BASE UNITS
frequency	Hz	hertz	---	$\text{s}^{-1}$
force	N	newton	---	$\text{kg}\cdot\text{m}\cdot\text{s}^{-2}$
pressure	Pa	pascal	$\text{N}/\text{m}^2$	$\text{kg}\cdot\text{m}^{-1}\cdot\text{s}^{-2}$
energy	J	joule	$\text{N}\cdot\text{m}$	$\text{kg}\cdot\text{m}^2\cdot\text{s}^{-2}$
power	W	watt	$\text{J}/\text{s}$	$\text{kg}\cdot\text{m}^2\cdot\text{s}^{-3}$
electric charge	C	coulomb	---	$\text{s}\cdot\text{A}$
voltage	V	volt	$\text{W}/\text{A}$	$\text{kg}\cdot\text{m}^2\cdot\text{s}^{-3}\cdot\text{A}^{-1}$
capacitance	F	farad	$\text{C}/\text{V}$	$\text{kg}^{-1}\cdot\text{m}^{-2}\cdot\text{s}^4\cdot\text{A}^2$
resistance	$\Omega$	ohm	$\text{V}/\text{A}$	$\text{kg}\cdot\text{m}^2\cdot\text{s}^{-3}\cdot\text{A}^{-2}$

# Dimensional Analysis

---

Power = Energy / Time

# Dimensional Analysis

---

Power = Energy / Time

- W

# Dimensional Analysis

---

Power = Energy / Time

- W
- J/s

# Dimensional Analysis

---

Power = Energy / Time

- W
- J/s
- N·m/s
- kg·m·s<sup>-2</sup>·m/s
- kg·m<sup>2</sup>·s<sup>-2</sup>/s
- kg·m<sup>2</sup>·s<sup>-3</sup>

# Dimensional Analysis

---

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

# Dimensional Analysis

---

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

```
// unit conversions  
static_assert(1_h == 3600_s);  
static_assert(1_km + 1_m == 1001_m);
```

# Dimensional Analysis

---

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

```
// unit conversions
static_assert(1_h == 3600_s);
static_assert(1_km + 1_m == 1001_m);
```

```
// dimension conversions
static_assert(1_km / 1_s == 1000_mps);
static_assert(2_kmph * 2_h == 4_km);
static_assert(2_km / 2_kmph == 1_h);

static_assert(1000 / 1_s == 1_kHz);

static_assert(10_km / 5_km == 2);
```

# SI prefixes

---

```
template<intmax_t Num, intmax_t Den = 1>
struct ratio;
```

# SI prefixes

```
template<intmax_t Num, intmax_t Den = 1>
struct ratio;
```

```
typedef ratio<1,          1000000000000000000> atto;
typedef ratio<1,          100000000000000000> femto;
typedef ratio<1,          10000000000000000> pico;
typedef ratio<1,          1000000000000000> nano;
typedef ratio<1,          1000000000000000> micro;
typedef ratio<1,          1000000000000000> milli;
typedef ratio<1,          1000000000000000> centi;
typedef ratio<1,          1000000000000000> deci;
typedef ratio<           10, 1> deca;
typedef ratio<           100, 1> hecto;
typedef ratio<           1000, 1> kilo;
typedef ratio<           1000000, 1> mega;
typedef ratio<           1000000000, 1> giga;
typedef ratio<           1000000000000, 1> tera;
typedef ratio<           1000000000000000, 1> peta;
typedef ratio<           1000000000000000000, 1> exa;
```

# More than one system of measurement

- United States customary units
- Imperial units
- ...

Unit	Divisions	SI Equivalent
Exact relationships shown in <b>boldface</b>		
<b>International</b>		
1 <i>point</i> (p)		352.777 778 µm
1 <i>pica</i> (P)	<b>12 p</b>	4.233 333 mm
1 <i>inch</i> (in or ")	<b>6 P/</b>	<b>25.4 mm</b>
1 <i>foot</i> (ft or ')	<b>12 in</b>	<b>0.304 8 m<sup>[9]</sup></b>
1 <i>yard</i> (yd)	<b>3 ft</b>	<b>0.914 4 m<sup>[9]</sup></b>
1 <i>mile</i> (mi)	<b>5 280 ft or 1 760 yd</b>	<b>1.609 344 km</b>
<b>US Survey</b>		
1 <i>link</i> (li)	$\frac{33}{50}$ ft or 7.92 in	0.201 116 8 m
1 (survey) foot (ft)	$\frac{1200}{3937}$ m	0.304 800 61 m <sup>[9]</sup>
1 <i>rod</i> (rd)	<b>25 li or 16.5 ft</b>	5.029 21 m
1 <i>chain</i> (ch)	<b>4 rd or 66 ft</b>	20.116 84 m
1 <i>furlong</i> (fur)	<b>10 ch</b>	201.168 4 m
1 survey (or statute) <i>mile</i> (mi)	<b>8 fur</b>	1.609 347 km <sup>[9]</sup>
1 <i>league</i> (lea)	<b>3 mi</b>	4.828 042 km
<b>International Nautical<sup>[9]</sup></b>		
1 <i>fathom</i> (ftm)	<b>2 yd</b>	<b>1.828 8 m</b>
1 <i>cable</i> (cb)	<b>120 ftm</b> or 1.091 fur	<b>219.456 m</b>
1 <i>nautical mile</i> (NM or nmi)	8.439 cb or 1.151 mi	<b>1.852 km</b>

# After Boost.Units

---

## BASE DIMENSION

- measurable entity of interest
- essentially a tag type and provide no dimensional analysis functionality by itself
- i.e. length [**L**], mass [**M**], time [**T**], etc.

# After Boost.Units

---

## BASE DIMENSION

- measurable entity of interest
- essentially a tag type and provide no dimensional analysis functionality by itself
- i.e. length [ $L$ ], mass [ $M$ ], time [ $T$ ], etc.

## DIMENSION

- a collection of zero or more base dimensions
- each potentially raised to a different rational power
- i.e. length =  $[L]^1$ , area =  $[L]^2$ , velocity =  $[L]^1/[T]^1$ , etc.

# After Boost.Units

---

## BASE UNIT

- a specific measure of a dimension
- conversions are defined using base units
- i.e.  $[L] \rightarrow m$  (meter)

# After Boost.Units

---

## BASE UNIT

- a specific measure of a dimension
- conversions are defined using base units
- i.e.  $[L] \rightarrow m$  (meter)

## UNIT

- a set of base units raised to rational exponents
- i.e.  $m^1, kg^1, m^1/s^2$ , etc.

# After Boost.Units

---

## SYSTEM

- a collection of base units representing all the measurable entities of interest for a specific problem

# After Boost.Units

---

## SYSTEM

- a collection of base units representing all the measurable entities of interest for a specific problem

## QUANTITY

- a concrete amount of a unit
- i.e. **10 meters**

## CURRENT STATE

### REVIEW OF EXISTING SOLUTIONS

# Toy example

---

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

# Toy example

---

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

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

# Toy example

---

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

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

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

# Toy example

---

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

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

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

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

# Toy example

---

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

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

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

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

# Toy example

---

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

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

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

- **Compile time safety** to make sure that the result is of a correct dimension
- Support for **multiple units and unit prefixes**
- **No runtime overhead**

# Toy example

---

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

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

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

- **Compile time safety** to make sure that the result is of a correct dimension
- Support for **multiple units and unit prefixes**
- **No runtime overhead**
- I/O output is out-of-scope for now (waiting for **std::format()**)

# Existing solutions

---

- **Boost.Units**

- authors: Matthias C. Schabel, Steven Watanabe
- [https://www.boost.org/doc/libs/1\\_69\\_0/doc/html/boost\\_units.html](https://www.boost.org/doc/libs/1_69_0/doc/html/boost_units.html)

# Existing solutions

---

- **Boost.Units**

- authors: Matthias C. Schabel, Steven Watanabe
  - [https://www.boost.org/doc/libs/1\\_69\\_0/doc/html/boost\\_units.html](https://www.boost.org/doc/libs/1_69_0/doc/html/boost_units.html)

- **Units**

- author: Nic Holthaus
  - <https://github.com/nholthaus/units>

# Existing solutions

---

- **Boost.Units**

- authors: Matthias C. Schabel, Steven Watanabe
  - [https://www.boost.org/doc/libs/1\\_69\\_0/doc/html/boost\\_units.html](https://www.boost.org/doc/libs/1_69_0/doc/html/boost_units.html)

- **Units**

- author: Nic Holthaus
  - <https://github.com/nholthaus/units>

- **std::chrono**

- author: Howard Hinnant

# Boost.Units: Toy example

---

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

# Boost.Units: Toy example

---

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

```
namespace bu = boost::units;
```

# Boost.Units: Toy example

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

```
namespace bu = boost::units;
```

```
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: Toy example

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

```
namespace bu = boost::units;
```

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

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

# Boost.Units: Toy example

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

```
namespace bu = boost::units;
```

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

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

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

# Boost.Units: Toy example

---

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

# Boost.Units: Toy example

---

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

```
const auto v = avg_speed(220 * bu::si::kilo * bu::si::meters, 2 * hours);
using kilometers_per_hour = bu::divide_typeof_helper<kilometer_base_unit::unit_type,
                                                    bu::metric::hour_base_unit::unit_type>::type;
bu::quantity<kilometers_per_hour> kmph(v);
std::cout << kmph.value() << " km/h\n";
```

# Boost.Units: Toy example

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

```
const auto v = avg_speed(220 * bu::si::kilo * bu::si::meters, 2 * hours);
using kilometers_per_hour = bu::divide_typeof_helper<kilometer_base_unit::unit_type,
                                                    bu::metric::hour_base_unit::unit_type>::type;
bu::quantity<kilometers_per_hour> kmph(v);
std::cout << kmph.value() << " km/h\n";
```

```
const auto v = avg_speed(140 * miles, 2 * hours);
using miles_per_hour = bu::divide_typeof_helper<bu::us::mile_base_unit::unit_type,
                                                    bu::metric::hour_base_unit::unit_type>::type;
bu::quantity<miles_per_hour> mph(v);
std::cout << mph.value() << " mph\n";
```

# Boost.Units: Toy example

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

```
const auto v = avg_speed(220 * bu::si::kilo * bu::si::meters, 2 * hours);
using kilometers_per_hour = bu::divide_typeof_helper<kilometer_base_unit::unit_type,
                                                    bu::metric::hour_base_unit::unit_type>::type;
bu::quantity<kilometers_per_hour> kmph(v);
std::cout << kmph.value() << " km/h\n";
```

```
const auto v = avg_speed(140 * miles, 2 * hours);
using miles_per_hour = bu::divide_typeof_helper<bu::us::mile_base_unit::unit_type,
                                                    bu::metric::hour_base_unit::unit_type>::type;
bu::quantity<miles_per_hour> mph(v);
std::cout << mph.value() << " mph\n";
```

Works, but runtime does unnecessary operations, and I may lose some bits of information while doing that

# Boost.Units: Toy example

---

- Too generic

```
template<typename Length, typename Time>
constexpr auto avg_speed(bu::quantity<Length> d, bu::quantity<Time> t)
{ return d / t; }
```

# Boost.Units: Toy example

---

- Too generic

```
template<typename Length, typename Time>
constexpr auto avg_speed(bu::quantity<Length> d, bu::quantity<Time> t)
{ return d / t; }
```

- Is it really a velocity dimension?

```
template<typename LengthSystem, typename Rep1, typename TimeSystem, typename Rep2>
constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>, Rep1> d,
                        bu::quantity<bu::unit<bu::time_dimension, TimeSystem>, Rep2> t)
{ return d / t; }
```

# Boost.Units: Toy example

- Too generic

```
template<typename Length, typename Time>
constexpr auto avg_speed(bu::quantity<Length> d, bu::quantity<Time> t)
{ return d / t; }
```

- Is it really a velocity dimension?

```
template<typename LengthSystem, typename Rep1, typename TimeSystem, typename Rep2>
constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>, Rep1> d,
                        bu::quantity<bu::unit<bu::time_dimension, TimeSystem>, Rep2> t)
{ return d / t; }
```

- Manually repeats built-in dimensional analysis logic

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

# Boost.Units: Toy example

---

```
template<typename LengthSystem, typename Rep1, typename TimeSystem, typename Rep2>
constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>, Rep1> d,
                        bu::quantity<bu::unit<bu::time_dimension, TimeSystem>, Rep2> t);
```

# Boost.Units: Toy example

---

```
template<typename LengthSystem, typename Rep1, typename TimeSystem, typename Rep2>
constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>, Rep1> d,
                        bu::quantity<bu::unit<bu::time_dimension, TimeSystem>, Rep2> t);
```

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

# Boost.Units: Toy example

---

```
template<typename LengthSystem, typename Rep1, typename TimeSystem, typename Rep2>
constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>, Rep1> d,
                        bu::quantity<bu::unit<bu::time_dimension, TimeSystem>, Rep2> t);
```

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

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

# Boost.Units: Toy example

---

```
template<typename LengthSystem, typename Rep1, typename TimeSystem, typename Rep2>
constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>, Rep1> d,
                        bu::quantity<bu::unit<bu::time_dimension, TimeSystem>, Rep2> t);
```

```
bu::quantity<length_kilometer> d = 220 * bu::si::kilo * bu::si::meters;
bu::quantity<time_hour> t = 2 * hours;
const auto kmph = avg_speed(d, t);
std::cout << kmph.value() << " km/h\n";
```

# Boost.Units: Toy example

---

```
template<typename LengthSystem, typename Rep1, typename TimeSystem, typename Rep2>
constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>, Rep1> d,
                        bu::quantity<bu::unit<bu::time_dimension, TimeSystem>, Rep2> t);
```

```
bu::quantity<length_kilometer> d = 220 * bu::si::kilo * bu::si::meters;
bu::quantity<time_hour> t = 2 * hours;
const auto kmph = avg_speed(d, t);
std::cout << kmph.value() << " km/h\n";
```

```
bu::quantity<length_mile> d = 140 * miles;
bu::quantity<time_hour> t = 2 * hours;
const auto mph = avg_speed(d, t);
std::cout << mph.value() << " mph\n";
```

# Boost.Units: Summary

---

## PROS

- The *widest adoption* thanks to Boost
- A wide range of *systems and base units*
- *High flexibility and extensibility*
- **constexpr** usage helps in compile-time
- **quantity** can use *any number-like type* for its representation

# Boost.Units: Summary

---

## PROS

- The *widest adoption* thanks to Boost
- A wide range of *systems and base units*
- *High flexibility and extensibility*
- **constexpr** usage helps in compile-time
- **quantity** can use *any number-like type* for its representation

## CONS

- *Pre-C++11* design
- Heavily relies on *macros* and *Boost.MPL*
- Domain and C++ *experts only*
  - poor compile-time error messages
  - no easy way to use non-SI units
  - spread over too many small headers (hard to compile a simple program)
  - designed around custom unit systems
- Not possible to explicitly construct a quantity of known unit from a plain value (even if no truncation occurs)

# Units: Toy example

---

```
#include "units.h"  
  
using namespace units::literals;
```

# Units: Toy example

---

```
#include "units.h"

using namespace units::literals;

template<typename Length, typename Time>
constexpr auto avg_speed(Length d, Time t)
{
    const auto v = d / t;
    return v;
}
```

# Units: Toy example

---

```
#include "units.h"

using namespace units::literals;

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

# Units: Toy example

```
#include "units.h"

using namespace units::literals;

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

- Not possible to define template arguments that will provide proper overload resolution because of unit nesting

```
using meter_t = units::unit_t<units::unit<std::ratio<1>, units::category::length_unit>>;
using kilometer_t = units::unit_t<units::unit<std::ratio<1000, 1>, meter_t, std::ratio<0, 1>, std::ratio<0, 1>>>;
```

# Units: Toy example

---

```
template<typename Length, typename Time>
constexpr auto avg_speed(Length d, Time t);
```

# Units: Toy example

---

```
template<typename Length, typename Time>
constexpr auto avg_speed(Length d, Time t);
```

```
const auto kmph = avg_speed(220_km, 2_hr);
std::cout << kmph.value() << " km/h\n";
```

# Units: Toy example

---

```
template<typename Length, typename Time>
constexpr auto avg_speed(Length d, Time t);
```

```
const auto kmph = avg_speed(220_km, 2_hr);
std::cout << kmph.value() << " km/h\n";
```

```
const auto mph = avg_speed(140_mi, 2_hr);
std::cout << mph.value() << " mph\n";
```

# Units: Toy example

---

```
template<typename Length, typename Time>
constexpr auto avg_speed(Length d, Time t);
```

```
const auto kmph = avg_speed(units::length::kilometer_t(220), units::time::hour_t(2));
std::cout << kmph.value() << " km/h\n";
```

```
const auto mph = avg_speed(units::length::mile_t(140), units::time::hour_t(2));
std::cout << mph.value() << " mph\n";
```

# Units: Toy example

---

```
template<typename Length, typename Time>
constexpr auto avg_speed(Length d, Time t);
```

```
units::length::kilometer_t d(220);
units::time::hour_t t(2);
const auto kmph = avg_speed(d, t);
std::cout << kmph.value() << " km/h\n";
```

```
units::length::mile_t d(140);
units::time::hour_t t(2);
const auto mph = avg_speed(d, t);
std::cout << mph.value() << " mph\n";
```

# Units: Toy example

```
template<typename Length, typename Time>
constexpr auto avg_speed(Length d, Time t);
```

```
units::length::kilometer_t d(220);
units::time::hour_t t(2);
const auto kmph = avg_speed(d, t);
std::cout << kmph.value() << " km/h\n";
```

```
units::length::mile_t d(140);
units::time::hour_t t(2);
const auto mph = avg_speed(d, t);
std::cout << mph.value() << " mph\n";
```

**meter** is a unit not a quantity!

-- Walter Brown

# Units: Summary

---

## PROS

- *Single header* file `units.h`
- The conversions between units are defined as  
*ratios at compile time*
- *UDL support*

# Units: Summary

---

## PROS

- *Single header* file `units.h`
- The conversions between units are defined as *ratios at compile time*
- *UDL support*

## CONS

- Not possible to *extend with own base units*
- Poor compile-time *error messages*
- No types that represent dimensions (*units only*)
- Mixing quantities with units
- Not easily suitable for *generic programming*

## ISSUES WITH CURRENT SOLUTIONS

# User experience: Compilation: Boost.Units

---

```
bu::quantity<bu::si::length> d = a * bu::si::kilo * bu::si::meters;
```

# User experience: Compilation: Boost.Units

```
bu::quantity<bu::si::length> d = a * bu::si::kilo * bu::si::meters;
```

GCC-8

```
error: conversion from ‘quantity<unit<[...],boost::units::heterogeneous_system<boost::units::heterogeneous_system_impl<boost::units::list<boost::units::heterogeneous_system_dim<boost::units::si::meter_base_unit, boost::units::static_rational<1> >, boost::units::dimensionless_type>, boost::units::list<boost::units::dim<boost::units::length_base_dimension, boost::units::static_rational<1> >, boost::units::dimensionless_type>, boost::units::list<boost::units::scale_list_dim<boost::units::scale<10, boost::units::static_rational<3> >>, boost::units::dimensionless_type> >>,[...]>,[...]>’ to non-scalar type ‘quantity<unit<[...], 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::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> >>>>>>>>>,[...]>,[...]>’ requested
    bu::quantity<bu::si::length> d = a * bu::si::kilo * bu::si::meters;
                                              ~~~~~^~~~~~
```

# User experience: Compilation: Boost.Units

---

```
bu::quantity<bu::si::length> d = a * bu::si::kilo * bu::si::meters;
```

## CLANG-7

```
error: no viable conversion from 'quantity<unit<[...], typename detail::multiply_systems<heterogeneous_system<heterogeneous_system_impl<dimensionless_type, dimensionless_type, list<scale_list_dim<scale<10, static_rational<3> >, dimensionless_type> >, typename detail::make_heterogeneous_system<list<dim<length_base_dimension, static_rational<1, 1> >, dimensionless_type>, list<meter_base_unit, list<scaled_base_unit<gram_base_unit, scale<10, static_rational<3> >, list<second_base_unit, list<ampere_base_unit, list<kelvin_base_unit, list<mole_base_unit, list<candela_base_unit, list<radian_base_unit, list<steradian_base_unit, dimensionless_type> >>>>>>>> >::type>::type>, [...]>' to 'quantity<unit<[...], si::system>, [...]>'  
    bu::quantity<bu::si::length> d = a * bu::si::kilo * bu::si::meters;  
                                   ^ ~~~~~~
```

# User experience: Compilation: Boost.Units

```
bu::quantity<bu::si::length> d = a * bu::si::kilo * bu::si::meters;
```

CLANG-7

```
error: no viable conversion from 'quantity<unit<[...], typename detail::multiply_systems<heterogeneous_system<heterogeneous_system_impl<dimensionless_type, dimensionless_type, list<scale_list_dim<scale<10, static_rational<3> >, dimensionless_type> >, typename detail::make_heterogeneous_system<list<dim<length_base_dimension, static_rational<1, 1> >, dimensionless_type>, list<meter_base_unit, list<scaled_base_unit<gram_base_unit, scale<10, static_rational<3> >, list<second_base_unit, list<ampere_base_unit, list<kelvin_base_unit, list<mole_base_unit, list<candela_base_unit, list<radian_base_unit, list<steradian_base_unit, dimensionless_type> >>>>>>>>::type>::type>, [...]>' to 'quantity<unit<[...], si::system>, [...]>'  
    bu::quantity<bu::si::length> d = a * bu::si::kilo * bu::si::meters;  
           ^ ~~~~~~
```

Can you guess what is wrong here?

# User experience: Compilation: Boost.Units

---

```
bu::quantity<bu::si::length> d(a * bu::si::kilo * bu::si::meters);
```

# User experience: Compilation: Boost.Units

---

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

# User experience: Compilation: Boost.Units

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

GCC-8

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

# User experience: Compilation: Boost.Units

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

## GCC-8 (CONTINUED)

```
...  
boost::units::quantity<boost::units::unit<boost::units::list<boost::units::dim<boost::units::length_base_dimension,  
boost::units::static_rational<1> >, boost::units::list<boost::units::dim<boost::units::time_base_dimension,  
boost::units::static_rational<1> >, boost::units::dimensionless_type> >, boost::units::homogeneous_system  
<boost::units::list<boost::units::si::meter_base_unit, boost::units::list<boost::units::scaled_base_unit  
<boost::units::cgs::gram_base_unit, boost::units::scale<10, boost::units::static_rational<3> > >,  
boost::units::list<boost::units::si::second_base_unit, boost::units::list<boost::units::si::ampere_base_unit,  
boost::units::list<boost::units::si::kelvin_base_unit, boost::units::list<boost::units::si::mole_base_unit,  
boost::units::list<boost::units::si::candela_base_unit, boost::units::list<boost::units::angle::radian_base_unit,  
boost::units::list<boost::units::angle::steradian_base_unit, boost::units::dimensionless_type> > > > > > > > > > > > ,  
void>, double>](t)' from 'quantity<unit<list<[...],list<dim<[...],static_rational<1>,[...]>>,[...],[...]>,[...]>'  
to 'quantity<unit<list<[...],list<dim<[...],static_rational<-1>,[...]>>,[...],[...]>,[...]>'  
return d * t;  
~~^~~
```

# User experience: Compilation: Boost.Units

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

## CLANG-7

```
error: no viable conversion from returned value of type 'quantity<unit<list<[...], list<dim<[...], static_rational<1, [...]>>, [...]>>, [...]>, [...]>' to function return type 'quantity<unit<list<[...], list<dim<[...], static_rational<-1, [...]>>, [...]>>, [...]>, [...]>'  
    return d * t;  
    ^~~~~~
```

# User experience: Compilation: Boost.Units

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

## CLANG-7

```
error: no viable conversion from returned value of type 'quantity<unit<list<[...], list<dim<[...], static_rational<1, [...]>>, [...]>>, [...]>, [...]>' to function return type 'quantity<unit<list<[...], list<dim<[...], static_rational<-1, [...]>>, [...]>>, [...]>, [...]>'  
    return d * t;  
    ^~~~~~
```

Sometimes shorter error message is not necessary better ;-)

# User experience: Compilation: Units

---

```
template<typename Length, typename Time>
constexpr auto avg_speed(Length d, Time t) { return d * t; }
```

# User experience: Compilation: Units

---

```
template<typename Length, typename Time>
constexpr auto avg_speed(Length d, Time t) { return d * t; }
```

GCC-8

```
error: static assertion failed: Units are not compatible.
 static_assert(traits::is_convertible_unit<UnitFrom, UnitTo>::value, "Units are not compatible.");
           ^~~~~~
```

# User experience: Compilation: Units

```
template<typename Length, typename Time>
constexpr auto avg_speed(Length d, Time t) { return d * t; }
```

GCC-8

```
error: static assertion failed: Units are not compatible.
static_assert(traits::is_convertible_unit<UnitFrom, UnitTo>::value, "Units are not compatible.");
^~~~~~
```

**static\_assert's** are often not the best solution

- do not influence the overload resolution process
- for some compilers do not provide enough context

# User experience: Compilation: Units

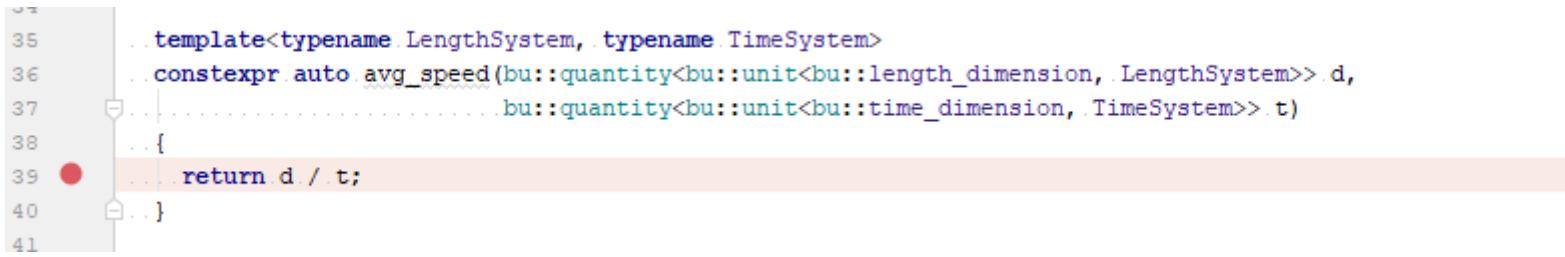
```
template<typename Length, typename Time>
constexpr auto avg_speed(Length d, Time t) { return d * t; }
```

## CLANG-7

```
error: static_assert failed due to requirement 'traits::is_convertible_unit<unit<ratio<3600000, 1>, base_unit<ratio<1, 1>, ratio<0, 1>, ratio<1, 1>, ratio<0, 1>, ratio<-1, 1>, ratio<0, 1>, ratio<0, 1>, ratio<0, 1>, ratio<0, 1>, ratio<0, 1>, ratio<0, 1>>::value' "Units are not compatible."
    static_assert(traits::is_convertible_unit<UnitFrom, UnitTo>::value, "Units are not compatible.");
^ ~~~~~
```

# User experience: Debugging: Boost.Units

---



A screenshot of a code editor or debugger interface showing a C++ file. A red circular breakpoint marker is positioned on the left margin next to line 39. The code on line 39 is highlighted with a light orange background. The code is as follows:

```
35     ...template<typename LengthSystem, typename TimeSystem>
36     ...constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>> d,
37     ...                                bu::quantity<bu::unit<bu::time_dimension, TimeSystem>> t)
38     ...
39     ...    return d / t;
40     ...
41 
```

# User experience: Debugging: Boost.Units

The screenshot shows a debugger interface with a code editor and a variables view.

**Code Editor:**

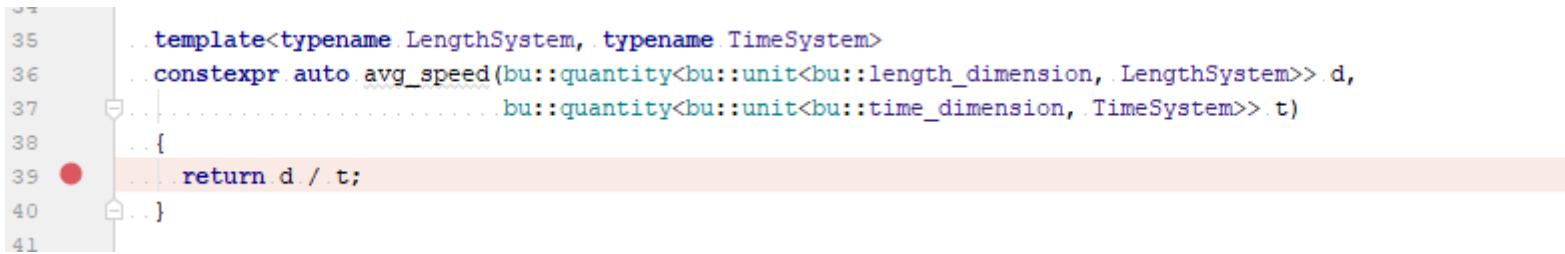
```
35     ...template<typename LengthSystem, typename TimeSystem>
36     ...constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>> d,
37     ...                                bu::quantity<bu::unit<bu::time_dimension, TimeSystem>> t)
38     ...
39     return d / t;
40 }
41 }
```

A red dot marks the current line of execution at line 39.

**Variables View:**

Variable	Type	Value
d	{boost::units::quantity<boost::units::unit, double>}	01 val_= {boost::units::quantity<boost::units::unit, double>::value_type} 220
t	{boost::units::quantity<boost::units::unit, double>}	01 val_= {boost::units::quantity<boost::units::unit, double>::value_type} 2

# User experience: Debugging: Boost.Units

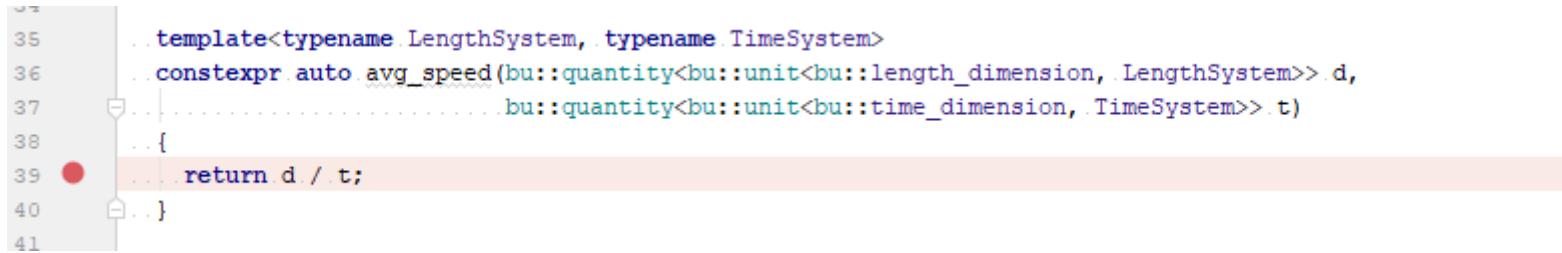


A screenshot of a debugger interface showing a code editor with C++ code. A red dot marks a breakpoint at line 39. The code is a template function for calculating average speed:

```
35     ..template<typename LengthSystem, typename TimeSystem>
36     ..constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>> d,
37     ..                           bu::quantity<bu::unit<bu::time_dimension, TimeSystem>> t)
38     ..{
39     ..    return d / t;
40     ..}
```

Breakpoint 1, (anonymous namespace)::avg\_speed<boost::units::heterogeneous\_system<boost::units::heterogeneous\_system\_impl<boost::units::list<boost::units::heterogeneous\_system\_dim<boost::units::si::meter\_base\_unit, boost::units::static\_rational<1> >, boost::units::dimensionless\_type>, boost::units::list<boost::units::dim<boost::units::length\_base\_dimension, boost::units::static\_rational<1> >, boost::units::dimensionless\_type>, boost::units::list<boost::units::scale\_list\_dim<boost::units::scale<10, boost::units::static\_rational<3> > >, boost::units::dimensionless\_type> > >, boost::units::heterogeneous\_system<boost::units::heterogeneous\_system\_impl<boost::units::list<boost::units::heterogeneous\_system\_dim<boost::units::scaled\_base\_unit<boost::units::si::second\_base\_unit, boost::units::scale<60, boost::units::static\_rational<2> > >, boost::units::static\_rational<1> >, boost::units::dimensionless\_type>, boost::units::list<boost::units::dim<boost::units::time\_base\_dimension, boost::units::static\_rational<1> >, boost::units::dimensionless\_type>, boost::units::dimensionless\_type> > > (d=..., t=...) at velocity\_2.cpp:39  
39 return d / t;

# User experience: Debugging: Boost.Units

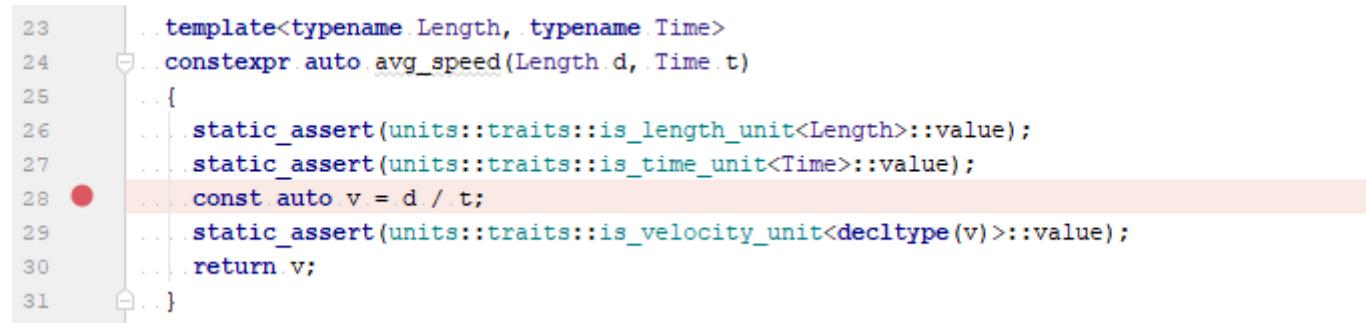


```
35     ...template<typename LengthSystem, typename TimeSystem>
36     ...constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>> d,
37     ...                                bu::quantity<bu::unit<bu::time_dimension, TimeSystem>> t)
38     ...
39     ...    return d / t;
40     ...
41 }
```

```
(gdb) ptype d
type = class boost::units::quantity<boost::units::unit<boost::units::list<boost::units::dim<boost::units::length_base_dimension,
boost::units::static_rational<1, 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, 1>, boost::units::dimensionless_type>,
boost::units::list<boost::units::dim<boost::units::length_base_dimension, boost::units::static_rational<1, 1>,
boost::units::dimensionless_type>, boost::units::list<boost::units::scale_list_dim<boost::units::scale<10, static_rational<3> > >
boost::units::dimensionless_type> > >, void>, double> [with Unit = boost::units::unit<boost::units::list<boost::units::dim
<boost::units::length_base_dimension, boost::units::static_rational<1, 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, 1>,
boost::units::dimensionless_type>, boost::units::list<boost::units::dim<boost::units::length_base_dimension,
boost::units::static_rational<1, 1>, boost::units::dimensionless_type>, boost::units::list<boost::units::scale_list_dim
<boost::units::scale<10, static_rational<3> > >, boost::units::dimensionless_type> > >, void>, Y = double] {
...
...
```

# User experience: Debugging: Units

---



A screenshot of a code editor or debugger interface showing a C++ function definition. The code is as follows:

```
23     . . . template<typename Length, typename Time>
24     . . . constexpr auto avg_speed(Length d, Time t)
25     . . . {
26     . . .     static_assert(units::traits::is_length_unit<Length>::value);
27     . . .     static_assert(units::traits::is_time_unit<Time>::value);
28     . . .     const auto v = d / t;
29     . . .     static_assert(units::traits::is_velocity_unit<decltype(v)>::value);
30     . . .     return v;
31     . . . }
```

The line 28, which contains the assignment `v = d / t;`, has a red circular breakpoint icon to its left. The entire line is highlighted with a light orange background, and the entire code block is enclosed in a light gray box.

# User experience: Debugging: Units

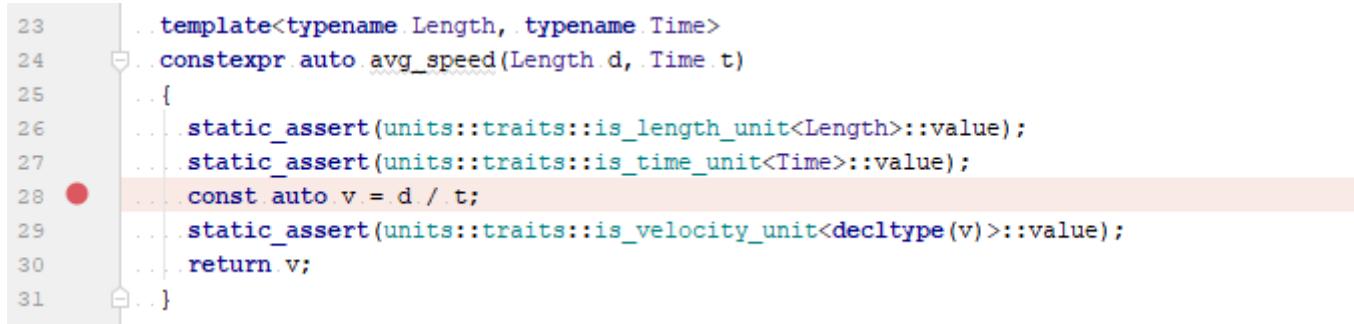
The screenshot shows a debugger interface with two main panes. The top pane displays a portion of C++ code:

```
23     .template<typename Length, typename Time>
24     constexpr auto avg_speed(Length d, Time t)
25     {
26         static_assert(units::traits::is_length_unit<Length>::value);
27         static_assert(units::traits::is_time_unit<Time>::value);
28         const auto v = d / t;
29         static_assert(units::traits::is_velocity_unit<decltype(v)>::value);
30         return v;
31     }
```

A red dot at line 28 indicates a breakpoint. The bottom pane is a 'Variables' view:

Variable	Type	Value
d	{units::unit_t<units::unit, double, units::linear_scale>}	
units::linear_scale<double>	{units::linear_scale<double>}	
m_value	{double}	220
units::detail::unit_t	{units::detail::unit_t}	
t	{units::unit_t<units::unit, double, units::linear_scale>}	
units::linear_scale<double>	{units::linear_scale<double>}	
m_value	{double}	2
units::detail::unit_t	{units::detail::unit_t}	

# User experience: Debugging: Units

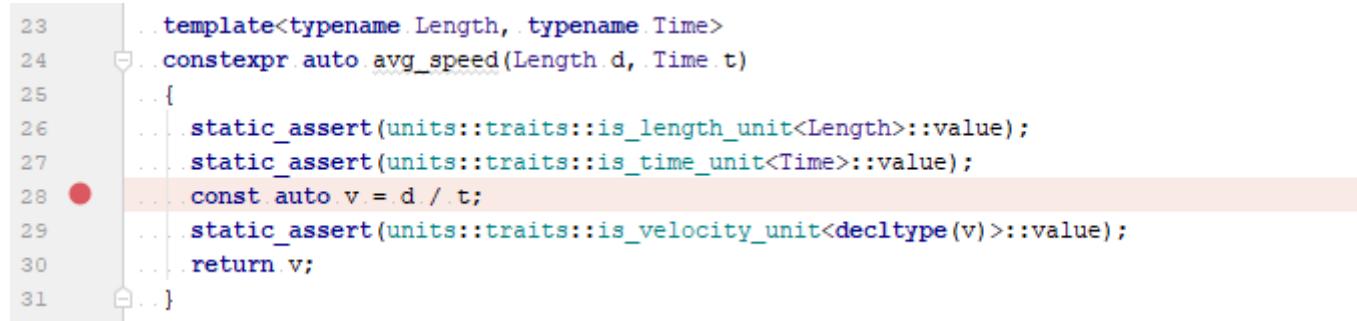


A screenshot of a debugger interface showing a C++ code editor. A red circular breakpoint marker is positioned at line 28. The code is a template function for calculating average speed:

```
23     .template<typename Length, typename Time>
24     constexpr auto avg_speed(Length d, Time t)
25     {
26         .static_assert(units::traits::is_length_unit<Length>::value);
27         .static_assert(units::traits::is_time_unit<Time>::value);
28         ● const auto v = d / t;
29         .static_assert(units::traits::is_velocity_unit<decltype(v)>::value);
30         .return v;
31     }
```

Breakpoint 1, (anonymous namespace)::avg\_speed<units::unit\_t<units::unit<std::ratio<1000, 1>, units::unit<std::ratio<1>, units::base\_unit<std::ratio<1> > >, std::ratio<0, 1>, std::ratio<0, 1> > >, units::unit\_t<units::unit<std::ratio<60>, units::unit<std::ratio<60>, units::unit<std::ratio<1>, units::base\_unit<std::ratio<0, 1>, std::ratio<0, 1>, std::ratio<1> > > > > > > > (d=..., t=...) at velocity.cpp:28  
28 const auto v = d / t;

# User experience: Debugging: Units



```
23     . template<typename Length, typename Time>
24     . constexpr auto avg_speed(Length d, Time t)
25     . {
26     .     static_assert(units::traits::is_length_unit<Length>::value);
27     .     static_assert(units::traits::is_time_unit<Time>::value);
28     .     const auto v = d / t;
29     .     static_assert(units::traits::is_velocity_unit<decltype(v)>::value);
30     .     return v;
31 }
```

```
(gdb) ptype d
type = class units::unit_t<units::unit<std::ratio<1000, 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> >, std::ratio<0, 1>, std::ratio<0, 1> >, double, units::linear_scale>
[with Units = units::unit<std::ratio<1000, 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> >, std::ratio<0, 1>, std::ratio<0, 1> >, T = double] : public units::linear_scale<T>,
private units::detail::_unit_t {
...
...
```

# Macros omnipresence: Boost.Units

---

```
BOOST_UNITS_DEFINE_CONVERSION_FACTOR(foot_base_unit, meter_base_unit, double, 0.3048);
```

```
BOOST_UNITS_DEFINE_CONVERSION_OFFSET(celsius_base_unit, fahrenheit_base_unit, double, 32.0);
```

```
BOOST_UNITS_DEFAULT_CONVERSION(my_unit_tag, SI::force);
```

```
BOOST_UNITS_DEFINE_CONVERSION_FACTOR_TEMPLATE((long N1)(long N2),  
    currency_base_unit<N1>,  
    currency_base_unit<N2>,  
    double, get_conversion_factor(N1, N2));
```

and more...

# Macros omnipresence: Units

```
#if !defined(DISABLE_PREDEFINED_UNITS) || defined(ENABLE_PREDEFINED_LENGTH_UNITS)
UNIT_ADD_WITH_METRIC_PREFIXES(length, meter, meters, m, unit<std::ratio<1>, units::category::length_unit>)
UNIT_ADD(length, foot, feet, ft, unit<std::ratio<381, 1250>, meters>)
UNIT_ADD(length, mil, mils, mil, unit<std::ratio<1000>, feet>)
UNIT_ADD(length, inch, inches, in, unit<std::ratio<1, 12>, feet>)
UNIT_ADD(length, mile, miles, mi, unit<std::ratio<5280>, feet>)
UNIT_ADD(length, nauticalMile, nauticalMiles, nmi, unit<std::ratio<1852>, meters>)
UNIT_ADD(length, astronomicalUnit, astronomicalUnits, au, unit<std::ratio<149597870700>, meters>)
UNIT_ADD(length, lightyear, lightyears, ly, unit<std::ratio<9460730472580800>, meters>)
UNIT_ADD(length, parsec, parsecs, pc, unit<std::ratio<648000>, astronomicalUnits, std::ratio<-1>>)
UNIT_ADD(length, angstrom, angstroms, angstrom, unit<std::ratio<1, 10>, nanometers>)
UNIT_ADD(length, cubit, cubits, cbt, unit<std::ratio<18>, inches>)
UNIT_ADD(length, fathom, fathoms, ftm, unit<std::ratio<6>, feet>)
UNIT_ADD(length, chain, chains, ch, unit<std::ratio<66>, feet>)
UNIT_ADD(length, furlong, furlongs, fur, unit<std::ratio<10>, chains>)
UNIT_ADD(length, hand, hands, hand, unit<std::ratio<4>, inches>)
UNIT_ADD(length, league, leagues, lea, unit<std::ratio<3>, miles>)
UNIT_ADD(length, nauticalLeague, nauticalLeagues, nl, unit<std::ratio<3>, nauticalMiles>)
UNIT_ADD(length, yard, yards, yd, unit<std::ratio<3>, feet>)

UNIT_ADD_CATEGORY_TRAIT(length)
#endif
```

# Extensibility

---

- Adding *derived dimensions is pretty easy* in all the libraries
- Adding *base dimensions is hard* or nearly impossible

# Extensibility

---

- Adding *derived dimensions is pretty easy* in all the libraries
- Adding *base dimensions is hard* or nearly impossible

## BOOST.UNITS

```
struct length_base_dimension : base_dimension<length_base_dimension, 1> {};
struct mass_base_dimension : base_dimension<mass_base_dimension, 2> {};
struct time_base_dimension : base_dimension<time_base_dimension, 3> {};
```

- Order is completely arbitrary as long as each tag has a *unique enumerable value*
- Non-unique ordinals are flagged as errors at compile-time
- *Negative ordinals are reserved* for use by the library
- Two independent libraries can easily choose the same ordinal (i.e. 1)

# Extensibility

---

- Adding *derived dimensions is pretty easy* in all the libraries
- Adding *base dimensions is hard* or nearly impossible

## UNITS

```
template<class Meter = detail::meter_ratio<0>,
         class Kilogram = std::ratio<0>,
         class Second = std::ratio<0>,
         class Radian = std::ratio<0>,
         class Ampere = std::ratio<0>,
         class Kelvin = std::ratio<0>,
         class Mole = std::ratio<0>,
         class Candela = std::ratio<0>,
         class Byte = std::ratio<0>>
struct base_unit;
```

- Requires *refactoring the engine, all existing predefined and users' unit types*

## **MY UNITS LIBRARY (WIP!!!)**

[HTTPS://GITHUB.COM/MPUSZ/UNITS](https://github.com/mpusz/units)

# Requirements

---

- Safety and performance

- strong types
- template metaprogramming
- **constexpr** all the things

# Requirements

---

- Safety and performance
  - strong types
  - template metaprogramming
  - **constexpr** all the things
- The best possible user experience
  - compiler errors
  - debugging

# Requirements

---

- Safety and performance
  - strong types
  - template metaprogramming
  - **constexpr** all the things
- The best possible user experience
  - compiler errors
  - debugging
- No macros in the user interface
- No external dependencies
- Easy extensibility

# Requirements

---

- Safety and performance
  - strong types
  - template metaprogramming
  - `constexpr` all the things
- The best possible user experience
  - compiler errors
  - debugging
- No macros in the user interface
- No external dependencies
- Easy extensibility
- Possibility to be standardized as a freestanding part of the C++ Standard Library

# Dimensions

---

- **units::dimension** - a *type-list-like type* that stores an *ordered list of exponents of one or more base dimensions*

```
template<typename... Exponents>
struct dimension;
```

# Dimensions

---

- **units::dimension** - a *type-list-like type* that stores an *ordered list of exponents of one or more base dimensions*

```
template<typename... Exponents>
struct dimension;
```

- **units::exp** - a *base dimension* and its *exponent* in a derived dimension

```
template<typename BaseDimension, int Value>
struct exp {
    using dimension = BaseDimension;
    static constexpr int value = Value;
};
```

# Dimensions

---

- **BaseDimension** is a *unique sortable compile-time value*

# Dimensions

---

- **BaseDimension** is a *unique sortable compile-time value*

```
template<int UniqueValue>
using dim_id = std::integral_constant<int, UniqueValue>;
```

# Dimensions

---

- **BaseDimension** is a *unique sortable compile-time value*

```
template<int UniqueValue>
using dim_id = std::integral_constant<int, UniqueValue>;
```

## EXAMPLE

```
struct base_dim_length : dim_id<0> {};
struct base_dim_mass : dim_id<1> {};
struct base_dim_time : dim_id<2> {};
```

# Dimensions

---

- **BaseDimension** is a *unique sortable compile-time value*

```
template<int UniqueValue>
using dim_id = std::integral_constant<int, UniqueValue>;
```

## EXAMPLE

```
struct base_dim_length : dim_id<0> {};
struct base_dim_mass : dim_id<1> {};
struct base_dim_time : dim_id<2> {};
```

The same problem with extensibility as with Boost.Units. If two users will select the same ID for their types than we have problems

# P0732 Class Types in Non-Type Template Parameters

---

- Allow *non-union class types* to appear in non-type template parameters
- Require that types used as such, and all of their bases and non-static data members recursively, *have a non-user-provided operator<=> returning* a type that is implicitly *convertible to std::strong\_equality*, and *contain no references*

# P0732 Class Types in Non-Type Template Parameters

---

- Allow *non-union class types* to appear in non-type template parameters
- Require that types used as such, and all of their bases and non-static data members recursively, *have a non-user-provided operator<=> returning* a type that is implicitly *convertible to std::strong\_equality*, and *contain no references*

```
template<fixed_string Id>
class entity { /* ... */ };

entity<"hello"> e;
```

# C++20 Exponent and Base Dimension

---

- **base\_dimension** can be either directly **fixed\_string** or a type that will include additional information (i.e. user's namespace name)

```
using base_dimension = fixed_string;
```

# C++20 Exponent and Base Dimension

---

- **base\_dimension** can be either directly **fixed\_string** or a type that will include additional information (i.e. user's namespace name)

```
using base_dimension = fixed_string;
```

- **exp** with C++20 support

```
template<base_dimension BaseDimension, int Value>
struct exp {
    static constexpr base_dimension dimension = BaseDimension;
    static constexpr int value = Value;
};
```

# C++20 Exponent and Base Dimension

---

- **base\_dimension** can be either directly **fixed\_string** or a type that will include additional information (i.e. user's namespace name)

```
using base_dimension = fixed_string;
```

- **exp** with C++20 support

```
template<base_dimension BaseDimension, int Value>
struct exp {
    static constexpr base_dimension dimension = BaseDimension;
    static constexpr int value = Value;
};
```

## EXAMPLE

```
inline constexpr base_dimension base_dim_length("length");
inline constexpr base_dimension base_dim_time("time");
```

# C++20 Exponent and Base Dimension

- **base\_dimension** can be either directly **fixed\_string** or a type that will include additional information (i.e. user's namespace name)

```
using base_dimension = fixed_string;
```

- **exp** with C++20 support

```
template<base_dimension BaseDimension, int Value>
struct exp {
    static constexpr base_dimension dimension = BaseDimension;
    static constexpr int value = Value;
};
```

Much easier to extend the library with new base dimension without identifier collisions between vendors

```
constexpr constexpr base_dimension base_dim_time( "time" ),
```

# Dimensions

---

- We can express **velocity** in the following way

```
dimension<exp<base_dim_length, 1>, exp<base_dim_time, -1>>
```

- improves *user experience*
- *as short as possible* template instantiations
- *easy to understand* by every engineer

# Dimensions

---

- We can express **velocity** in the following way

```
dimension<exp<base_dim_length, 1>, exp<base_dim_time, -1>>
```

- improves *user experience*
- *as short as possible* template instantiations
- *easy to understand* by every engineer

## PROBLEM

- The same type expected for both operations

```
constexpr auto v1 = 1_m / 1_s;
constexpr auto v2 = 2 / 2_s * 1_m;

static_assert(std::is_same<decltype(v1), decltype(v2)>);
```

# Dimensions: `make_dimension` factory helper

```
template<typename... Exponents>
struct make_dimension {
    using type = /* unspecified */;
};

template<typename... Exponents>
using make_dimension_t = make_dimension<Exponents...>::type;
```

- Provides *unique ordering* for contained base dimensions
- *Aggregates* two arguments of the same base dimension but *different exponents*
- *Eliminates two arguments* of the same base dimension and *with opposite equal exponents*

# Dimensions: `make_dimension` factory helper

```
template<typename... Exponents>
struct make_dimension {
    using type = /* unspecified */;
};

template<typename... Exponents>
using make_dimension_t = make_dimension<Exponents...>::type;
```

- Provides *unique ordering* for contained base dimensions
- *Aggregates* two arguments of the same base dimension but *different exponents*
- *Eliminates two arguments* of the same base dimension and *with opposite equal exponents*

## EXAMPLE

- To form a velocity

```
make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>
```

# Units

---

- **unit** - the unit of a specific physical dimension

```
template<typename Dimension, typename Ratio>
struct unit {
    using dimension = Dimension;
    using ratio = Ratio;
};
```

# Units

---

- **unit** - the unit of a specific physical dimension

```
template<typename Dimension, typename Ratio>
struct unit {
    using dimension = Dimension;
    using ratio = Ratio;
};
```

## EXAMPLE

To express **meter**

```
unit<dimension_length, std::ratio<1>>
```

# Quantities

---

- **quantity** - the *amount of a specific dimension* expressed *in a specific unit* of that dimension

```
template<typename Dimension, typename Unit, typename Rep>
class quantity;
```

# Quantities

---

- **quantity** - the *amount of a specific dimension* expressed *in a specific unit* of that dimension

```
template<typename Dimension, typename Unit, typename Rep>
class quantity;
```

- Interface similar to *std::chrono::duration + additional member functions*
  - multiplication of 2 quantities with different dimensions

```
1_kmph * 1_h == 1_km
```

- division of 2 quantities of different dimensions

```
1_km / 1_h == 1_kmph
```

- division of a scalar with a quantity

```
1 / 1_s == 1_Hz
```

# Quantities

---

- **quantity** - the *amount of a specific dimension* expressed *in a specific unit* of that dimension

```
template<typename Dimension, typename Unit, typename Rep>
class quantity;
```

- Interface similar to *std::chrono::duration + additional member functions*
  - quantity value equal to 1 expressed in a specific Rep

```
[[nodiscard]] static constexpr quantity one() noexcept;
```

# Quantities

---

- **quantity** - the *amount of a specific dimension* expressed *in a specific unit* of that dimension

```
template<typename Dimension, typename Unit, typename Rep>
class quantity;
```

## EXAMPLE

```
template<typename Unit = meter_per_second, typename Rep = double>
using velocity = quantity<dimension_velocity, Unit, Rep>;
```

# Struggling with the user experience

---

- 1 Generic programming
- 2 Compile-time errors
- 3 Debugging

# Do you remember that?

## BOOST.UNITS

- Is it really a velocity dimension?

```
template<typename LengthSystem, typename Rep1, typename TimeSystem, typename Rep2>
constexpr auto avg_speed(bu::quantity<bu::unit<bu::length_dimension, LengthSystem>, Rep1> d,
                        bu::quantity<bu::unit<bu::time_dimension, TimeSystem>, Rep2> t)
{ return d / t; }
```

## UNITS

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

- Not possible to define template arguments that will provide proper overload resolution

# C++ Concepts to the rescue

---

## BOOST.UNITS

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

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

# C++ Concepts to the rescue

---

## BOOST.UNITS

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

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

```
template<typename Quantity>
concept Length = QuantityOf<Quantity, bu::length_dimension>;
template<typename Quantity>
concept Time = QuantityOf<Quantity, bu::time_dimension>;
template<typename Quantity>
concept Velocity = QuantityOf<Quantity, bu::velocity_dimension>;
```

# C++ Concepts to the rescue

## BOOST.UNITS

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

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

```
template<typename Quantity>
concept Length = QuantityOf<Quantity, bu::length_dimension>;
template<typename Quantity>
concept Time = QuantityOf<Quantity, bu::time_dimension>;
template<typename Quantity>
concept Velocity = QuantityOf<Quantity, bu::velocity_dimension>;
```

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

# C++ Concepts to the rescue

---

## UNITS

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

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

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

# C++ Concepts to the rescue

## UNITS

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

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

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

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

# C++ Concepts to the rescue

## UNITS

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

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

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

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

Concepts can be used in places where regular template argument deduction does not work (i.e. return types, class template parameters, etc).

I lied a bit ;-)

---

All template types are heavily embraced with concepts

# I lied a bit ;-)

---

All template types are heavily embraced with concepts

## UNITS ENGINE CONCEPTS

- **TypeList**
- **Number**
- **Ratio**
- **Exponent**
- **Dimension**
- **Unit**
- **Quantity**

## All template types are heavily embraced with concepts

### UNITS ENGINE CONCEPTS

- **TypeList**
- **Number**
- **Ratio**
- **Exponent**
- **Dimension**
- **Unit**
- **Quantity**

### PREDEFINED QUANTITIES CONCEPTS

- **Length**
- **Time**
- **Frequency**
- **Velocity**
- ...

```
template<typename T>
concept Velocity = Quantity<T> &&
    std::Same<typename T::dimension, dimension_velocity>;
```

# Type aliases are great (but not for users)

---

- Velocity is one of the simplest derived dimensions one can imagine

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
```

# Type aliases are great (but not for users)

---

- Velocity is one of the simplest derived dimensions one can imagine

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
```

- Type *aliases names are lost* quickly during compilation process

# Type aliases are great (but not for users)

---

- Velocity is one of the simplest derived dimensions one can imagine

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
```

- Type *aliases names are lost* quickly during compilation process
- As a result user gets **huge types in error messages**

[with D = units::quantity<units::dimension<units::exp<units::base\_dim\_length, 1>, units::exp<units::base\_dim\_time, -1> >,  
units::unit<units::dimension<units::exp<units::base\_dim\_length, 1>, units::exp<units::base\_dim\_time, -1> >,  
std::ratio<1000, 3600> >, long long int>]

# Type aliases are great (but not for users)

---

- Velocity is one of the simplest derived dimensions one can imagine

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
```

- Type *aliases names are lost* quickly during compilation process
- As a result user gets **huge types in error messages**

[with D = units::quantity<units::dimension<units::exp<units::base\_dim\_length, 1>, units::exp<units::base\_dim\_time, -1> >,  
units::unit<units::dimension<units::exp<units::base\_dim\_length, 1>, units::exp<units::base\_dim\_time, -1> >,  
std::ratio<1000, 3600> >, long long int>]

It is a pity that we still do not have strong typedef's in the C++ language :-(

# Inheritance to the rescue

---

```
struct dimension_velocity : make_dimension_t<exp<base_dim_length, 1>,  
                           exp<base_dim_time, -1>> {};
```

- We get *strong types* that do not vanish during compilation process

# Inheritance to the rescue

---

```
struct dimension_velocity : make_dimension_t<exp<base_dim_length, 1>,
                                         exp<base_dim_time, -1>> {};
```

- We get *strong types* that do not vanish during compilation process
- *Easily applicable only to simple classes* because of problems with
  - constructors
  - assignment operators
  - comparison operators
  - ...

# Inheritance to the rescue

---

```
struct dimension_velocity : make_dimension_t<exp<base_dim_length, 1>,
                                         exp<base_dim_time, -1>> {};
```

- We get *strong types* that do not vanish during compilation process
- *Easily applicable only to simple classes* because of problems with
  - constructors
  - assignment operators
  - comparison operators
  - ...
- *CRTP could help* but it complicates the design

# Inheritance to the rescue

```
struct dimension_velocity : make_dimension_t<exp<base_dim_length, 1>,
                                         exp<base_dim_time, -1>> {};
```

- We get *strong types* that do not vanish during compilation process
- *Easily applicable only to simple classes* because of problems with
  - constructors
  - assignment operators
  - comparison operators
  - ...
- *CRTP could help* but it complicates the design
- Anyway **velocity** *should be considered a helper alias* for **quantity** rather than a strong type

```
template<Unit U = meter_per_second, Number Rep = double>
using velocity = quantity<dimension_velocity, U, Rep>;
```

# Upcasting problem

---

```
struct dimension_velocity : make_dimension_t<exp<base_dim_length, 1>,
                                         exp<base_dim_time, -1>> {};  
  
template<Unit U = meter_per_second, Number Rep = double>  
using velocity = quantity<dimension_velocity, U, Rep>;
```

```
Velocity auto v = 10_m / 2_s;
```

# Upcasting problem

```
struct dimension_velocity : make_dimension_t<exp<base_dim_length, 1>,
                                         exp<base_dim_time, -1>> {};  
  
template<Unit U = meter_per_second, Number Rep = double>  
using velocity = quantity<dimension_velocity, U, Rep>;  
  
Velocity auto v = 10_m / 2_s;
```

How to form **dimension\_velocity** class from division of **dimension\_length** by **dimension\_time**?

# P0887 The identity metafunction

---

```
template<typename T>
struct type_identity { using type = T; };

template<typename T>
using type_identity_t = type_identity<T>::type;
```

# Upcasting facility

---

## BASE CLASS

```
template<typename BaseType>
struct upcast_base {
    using base_type = BaseType;
};
```

```
template<Exponent... Es>
struct dimension : upcast_base<dimension<Es...>> {};
```

# Upcasting facility

---

## BASE CLASS

```
template<typename BaseType>
struct upcast_base {
    using base_type = BaseType;
};
```

```
template<Exponent... Es>
struct dimension : upcast_base<dimension<Es...>> {};
```

## HELPER ALIASES

```
template<Upcastable T>
using upcast_from = T::base_type;

template<typename T>
using upcast_to = std::type_identity<T>;
```

# Upcasting facility

---

## UPCASTING TRAITS

```
template<typename T>
struct upcasting_traits : upcast_to<T> {};  
  
template<typename T>
using upcasting_traits_t = upcasting_traits<T>::type;
```

# Upcasting facility

---

```
template<Exponent... Es>
struct dimension : upcast_base<dimension<Es...>> {};
```

```
struct dimension_velocity : make_dimension_t<exp<base_dim_length, 1>,
                                         exp<base_dim_time, -1>> {};
```

# Upcasting facility

---

```
template<Exponent... Es>
struct dimension : upcast_base<dimension<Es...>> {};
```

```
struct dimension_velocity : make_dimension_t<exp<base_dim_length, 1>,
                                         exp<base_dim_time, -1>> {};
```

```
template<>
struct upcasting_traits<upcast_from<dimension_velocity>> : upcast_to<dimension_velocity> {};
```

# Upcasting facility

```
template<Exponent... Es>
struct dimension : upcast_base<dimension<Es...>> {};
```

```
struct dimension_velocity : make_dimension_t<exp<base_dim_length, 1>,
                                         exp<base_dim_time, -1>> {};
```

```
template<>
struct upcasting_traits<upcast_from<dimension_velocity>> : upcast_to<dimension_velocity> {};
```

Help needed: How to automate the last line?

# Upcasting facility

---

## BASE UNIT

```
struct meter : unit<dimension_length, std::ratio<1>> {};
template<>
struct upcasting_traits<upcast_from<meter>> : upcast_to<meter> {};
```

# Upcasting facility

---

## BASE UNIT

```
struct meter : unit<dimension_length, std::ratio<1>> {};
template<>
struct upcasting_traits<upcast_from<meter>> : upcast_to<meter> {};
```

## UNIT WITH PREFIX

```
struct kilometer : kilo<meter> {};
template<>
struct upcasting_traits<upcast_from<kilometer>> : upcast_to<kilometer> {};
```

# Upcasting facility

## BASE UNIT

```
struct meter : unit<dimension_length, std::ratio<1>> {};
template<>
struct upcasting_traits<upcast_from<meter>> : upcast_to<meter> {};
```

## UNIT WITH PREFIX

```
struct kilometer : kilo<meter> {};
template<>
struct upcasting_traits<upcast_from<kilometer>> : upcast_to<kilometer> {};
```

## DERIVED UNIT

```
struct kilometer_per_hour : derived_unit<dimension_velocity, kilometer, hour> {};
template<>
struct upcasting_traits<upcast_from<kilometer_per_hour>> : upcast_to<kilometer_per_hour> {};
```

# Upcasting facility

---

## BEFORE

```
[with D = units::quantity<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, -1> >,  
units::unit<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, -1> >,  
std::ratio<1000, 3600>, long long int>]
```

# Upcasting facility

---

## BEFORE

```
[with D = units::quantity<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, -1> >,  
units::unit<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, -1> >,  
std::ratio<1000, 3600>, long long int>]
```

## AFTER

```
[with D = units::quantity<units::dimension_velocity, units::kilometer_per_hour, long long int>
```

# Upcasting facility

---

## BEFORE

```
[with D = units::quantity<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, -1> >,  
units::unit<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, -1> >,  
std::ratio<1000, 3600>, long long int>]
```

## AFTER

```
[with D = units::quantity<units::dimension_velocity, units::kilometer_per_hour, long long int>
```

## POSSIBLE WITH A DESIGN COST

```
[with D = units::velocity<units::kilometer_per_hour, long long int>
```

# Upcasting facility

---

## BEFORE

```
[with D = units::quantity<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, -1> >,  
units::unit<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, -1> >,  
std::ratio<1000, 3600>, long long int>]
```

## AFTER

```
[with D = units::quantity<units::dimension_velocity, units::kilometer_per_hour, long long int>
```

## POSSIBLE WITH A DESIGN COST

```
[with D = units::velocity<units::kilometer_per_hour, long long int>
```

Do we want the last option with a cost of CRTP usage in **quantity**?

# C++17 CTAD (Class Template Argument Deduction)

---

## CLASS TEMPLATE

```
template<class T, class Allocator = std::allocator<T>>
class vector;
```

# C++17 CTAD (Class Template Argument Deduction)

## CLASS TEMPLATE

```
template<class T, class Allocator = std::allocator<T>>
class vector;
```

```
vector v = {1, 2, 3};
vector v2(cont.begin(), cont.end());
```

# C++17 CTAD (Class Template Argument Deduction)

## CLASS TEMPLATE

```
template<class T, class Allocator = std::allocator<T>>
class vector;
```

```
vector v = {1, 2, 3};
vector v2(cont.begin(), cont.end());
```

## ALIAS TEMPLATE

```
namespace pmr {
    template<class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

# C++17 CTAD (Class Template Argument Deduction)

## CLASS TEMPLATE

```
template<class T, class Allocator = std::allocator<T>>
class vector;
```

```
vector v = {1, 2, 3};
vector v2(cont.begin(), cont.end());
```

## ALIAS TEMPLATE

```
namespace pmr {
    template<class T>
    using vector = std::vector<T, std::pmr::polymorphic_allocator<T>>;
}
```

```
pmr::vector<int> v = {1, 2, 3};
pmr::vector<decltype(cont)::value_type> v2(cont.begin(), cont.end());
```

# P1021 Filling holes in Class Template Argument Deduction

---

## ALIAS TEMPLATE

- Adds CTAD support for
  - aggregate templates
  - type aliases
  - inherited constructors

```
pmr::vector v = {1, 2, 3};  
pmr::vector v2(cont.begin(), cont.end());
```

# C++20 CTAD in Units

---

```
template<Dimension D, Unit U, Number Rep> class quantity;
template<Dimension D, Unit U, Number Rep> quantity(Rep r) -> quantity<D, U, Rep>;
  
template<Unit U = meter, Number Rep = double>
using length = quantity<dimension_length, U, Rep>;
```

# C++20 CTAD in Units

```
template<Dimension D, Unit U, Number Rep> class quantity;
template<Dimension D, Unit U, Number Rep> quantity(Rep r) -> quantity<D, U, Rep>;
template<Unit U = meter, Number Rep = double>
using length = quantity<dimension_length, U, Rep>;
```

```
units::length d1(3);                                // OK -> quantity<dimension_length, meter, int>
units::length d2(3.14);                             // OK -> quantity<dimension_length, meter, double>
units::length<units::mile> d3(3);                  // FAIL -> quantity<dimension_length, mile, double>
units::length<units::mile> d4(3.14);                // OK -> quantity<dimension_length, mile, double>
units::length<units::mile, float> d5(3.14);          // OK -> quantity<dimension_length, mile, float>
```

# C++20 CTAD in Units

```
template<Dimension D, Unit U, Number Rep> class quantity;
template<Dimension D, Unit U, Number Rep> quantity(Rep r) -> quantity<D, U, Rep>;
template<Unit U = meter, Number Rep = double>
using length = quantity<dimension_length, U, Rep>;
```

```
units::length d1(3);                                // OK -> quantity<dimension_length, meter, int>
units::length d2(3.14);                             // OK -> quantity<dimension_length, meter, double>
units::length<units::mile> d3(3);                  // FAIL -> quantity<dimension_length, mile, double>
units::length<units::mile> d4(3.14);                // OK -> quantity<dimension_length, mile, double>
units::length<units::mile, float> d5(3.14);          // OK -> quantity<dimension_length, mile, float>
```

## WORKAROUND THAT I WOULD PREFER NOT TO DO ("METER IS A UNIT NOT A QUANTITY! ")

```
template<Number Rep = double>
using miles = length<mile, Rep>;
```

```
units::miles d3(3);                                // OK -> quantity<dimension_length, mile, int>
```

# Contracts

---

```
template<Dimension D, Unit U, Number Rep>
    requires std::Same<D, typename U::dimension>
class quantity {
public:
    template<Dimension D, Unit U1, Number Rep1, Unit U2, Number Rep2>
    [[nodiscard]] std::common_type_t<Rep1, Rep2>
    constexpr operator/(const quantity<D, U1, Rep1>& lhs,
                        const quantity<D, U2, Rep2>& rhs)
    {
        Expects(rhs != quantity<D, U2, Rep2>(0));
        // ...
    }
};
```

# Contracts

---

```
template<Dimension D, Unit U, Number Rep>
    requires std::Same<D, typename U::dimension>
class quantity {
public:
    template<Dimension D, Unit U1, Number Rep1, Unit U2, Number Rep2>
    [[nodiscard]] std::common_type_t<Rep1, Rep2>
    constexpr operator/(const quantity<D, U1, Rep1>& lhs,
                        const quantity<D, U2, Rep2>& rhs)
    {
        Expects(rhs != quantity<D, U2, Rep2>(0));
        // ...
    }
};
```

error: macro "Expects" passed 3 arguments, but takes just 1

# Contracts

---

```
template<Dimension D, Unit U, Number Rep>
    requires std::Same<D, typename U::dimension>
class quantity {
public:
    template<Dimension D, Unit U1, Number Rep1, Unit U2, Number Rep2>
    [[nodiscard]] std::common_type_t<Rep1, Rep2>
    constexpr operator/(const quantity<D, U1, Rep1>& lhs,
                        const quantity<D, U2, Rep2>& rhs)
    {
        using rhs_type = quantity<D, U2, Rep2>;
        Requires(rhs != rhs_type(0));
        // ...
    }
};
```

# Contracts

---

```
template<Dimension D, Unit U, Number Rep>
    requires std::Same<D, typename U::dimension>
class quantity {
public:
    template<Dimension D, Unit U1, Number Rep1, Unit U2, Number Rep2>
    [[nodiscard]] std::common_type_t<Rep1, Rep2>
    constexpr operator/(const quantity<D, U1, Rep1>& lhs,
                        const quantity<D, U2, Rep2>& rhs)
    {
        Expects(rhs != std::remove_cvref_t<decltype(rhs)>{0});
        // ...
    }
};
```

# Contracts

```
template<Dimension D, Unit U, Number Rep>
    requires std::Same<D, typename U::dimension>
class quantity {
public:
    template<Dimension D, Unit U1, Number Rep1, Unit U2, Number Rep2>
    [[nodiscard]] std::common_type_t<Rep1, Rep2>
    constexpr operator/(const quantity<D, U1, Rep1>& lhs,
                        const quantity<D, U2, Rep2>& rhs)
    {
        Expects(rhs != std::remove_cvref_t<decltype(rhs)>{0});
        // ...
    }
};
```

Still not the best solution:

- usage of a macro in a header file (possible ODR issue)
- not a part of a function signature

# C++20 Contracts

---

```
template<Dimension D, Unit U, Number Rep>
    requires std::Same<D, typename U::dimension>
class quantity {
public:
    template<Dimension D, Unit U1, Number Rep1, Unit U2, Number Rep2>
    [[nodiscard]] std::common_type_t<Rep1, Rep2>
    constexpr operator/(const quantity<D, U1, Rep1>& lhs,
                        const quantity<D, U2, Rep2>& rhs) [[expects: rhs != quantity<D, U2, Rep2>(0)]]
    {
        // ...
    }
};
```

# Toy example

---

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

# Toy example

---

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

```
const auto kmph = avg_speed(220_km, 2_h);
std::cout << kmph.count() << " km/h\n";
```

```
const auto mph = avg_speed(140_mi, 2_h);
std::cout << mph.count() << " mph\n";
```

# Toy example

---

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

```
const auto kmph = avg_speed(units::length<units::kilometer>(220), units::time<units::hour>(2));
std::cout << kmph.count() << " km/h\n";
```

```
const auto mph = avg_speed(units::length<units::mile>(140), units::time<units::hour>(2));
std::cout << mph.count() << " mph\n";
```

# Toy example

---

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

```
units::length<units::kilometer> d(220);
units::time<units::hour> t(2);
const auto kmph = avg_speed(d, t);
std::cout << kmph.count() << " km/h\n";
```

```
units::length<units::mile> d(140);
units::time<units::hour> t(2);
const auto mph = avg_speed(d, t);
std::cout << mph.count() << " mph\n";
```

# User experience: Compilation

---

```
constexpr units::velocity<units::kilometer_per_hour> avg_speed(units::Length auto d, units::Time auto t)
{ return d * t; }
```

# User experience: Compilation

```
constexpr units::velocity<units::kilometer_per_hour> avg_speed(units::Length auto d, units::Time auto t)
{ return d * t; }
```

GCC-8

```
error: conversion from ‘quantity<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, 1> >, units::unit<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, 1> >, std::ratio<3600000, 1> > , [...]’ to non-scalar type ‘quantity<units::dimension_velocity,units::kilometer_per_hour,[...]>’
requested
```

# User experience: Compilation

```
constexpr units::velocity<units::kilometer_per_hour> avg_speed(units::Length auto d, units::Time auto t)
{ return d * t; }
```

GCC-8

```
error: conversion from 'quantity<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, 1> >,  
units::unit<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, 1> >,  
std::ratio<3600000, 1> > , [...]>' to non-scalar type 'quantity<units::dimension_velocity,units::kilometer_per_hour,[...]>'  
requested
```

Repeating of broken dimension type is unfortunate here, but it actually makes some code (i.e. using CTAD) easier to understand.  
Design decision tradeoff still to decide...

# User experience: Compilation after possible refactoring

---

```
constexpr units::quantity<units::kilometer_per_hour> avg_speed(units::Length auto d, units::Time auto t)
{ return d * t; }
```

# User experience: Compilation after possible refactoring

```
constexpr units::quantity<units::kilometer_per_hour> avg_speed(units::Length auto d, units::Time auto t)
{ return d * t; }
```

GCC-8

```
error: conversion from ‘quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1>,
units::exp<units::base_dim_time, 1> >, std::ratio<3600000, 1> > , [...]’ to non-scalar type
‘quantity<units::kilometer_per_hour,[...]>’ requested
```

# User experience: Compilation after possible refactoring

```
constexpr units::quantity<units::kilometer_per_hour> avg_speed(units::Length auto d, units::Time auto t)
{ return d * t; }
```

GCC-8

```
error: conversion from ‘quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1>,
units::exp<units::base_dim_time, 1> >, std::ratio<3600000, 1> >,[...]>’ to non-scalar type
‘quantity<units::kilometer_per_hour,[...]>’ requested
```

BEFORE

```
template<Dimension D, Unit U, Number Rep>
class quantity;

template<Unit U = meter, Number Rep = double>
using length = quantity<dimension_length, U, Rep>;
```

```
units::length d(3); // 3 meters
```

```
void foo(Quantity auto q) {
    units::length d(q); // Yay, we are working with length
}
```

AFTER

```
template<Unit U, Number Rep = double>
class quantity;
```

```
units::quantity d(3); // Huh???
```

```
void foo(Quantity auto q) {
    units::quantity d(q); // Huh???
}
```

# User experience: Compilation

---

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

# User experience: Compilation

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

## GCC-8 (FULL ERROR LOG)

```
velocity.cpp: In instantiation of ‘constexpr units::Velocity {anonymous}::avg_speed(D, T)
[with D = units::quantity<units::dimension_length, units::kilometer, double>;
T = units::quantity<units::dimension_time, units::hour, double>’:
/mnt/c/repos/units_compare/src/mpusz/velocity.cpp:23:37:   required from here
velocity.cpp:12:16:error: placeholder constraints not satisfied
    return d * t;
           ^
```

```
include/units/si/velocity.h:47:16: note: within ‘template<class T> concept const bool units::Velocity<T>
[with T = units::quantity<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, 1> >,
units::unit<units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, 1> >,
std::ratio<3600000, 1> >, double>’
```

```
concept Velocity = Quantity<T> && std::Same<typename T::dimension, dimension_velocity>;
           ^~~~~~
```

```
include/stl2/detail/concepts/core.hpp:37:15: note: within ‘template<class T, class U> concept const bool std::v1::Same<T, U>
[with T = units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, 1> >;
U = units::dimension_velocity’
```

```
META_CONCEPT Same = meta::Same<T, U> && meta::Same<U, T>;
           ^~~~
```

...

# User experience: Compilation

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

## GCC-8 (FULL ERROR LOG - CONTINUED)

```
...
include/meta/meta_fwd.hpp:206:18: note: within ‘template<class T, class U> concept const bool meta::Same<T, U>
[with T = units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, 1> >;
U = units::dimension_velocity]’
META_CONCEPT Same =
    ^~~~
include/meta/meta_fwd.hpp:206:18: note: ‘meta::detail::bool_’ evaluated to false
include/meta/meta_fwd.hpp:206:18: note: within ‘template<class T, class U> concept const bool meta::Same<T, U>
[with T = units::dimension_velocity;
U = units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, 1> >]’
include/meta/meta_fwd.hpp:206:18: note: ‘meta::detail::bool_’ evaluated to false
```

# User experience: Compilation

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

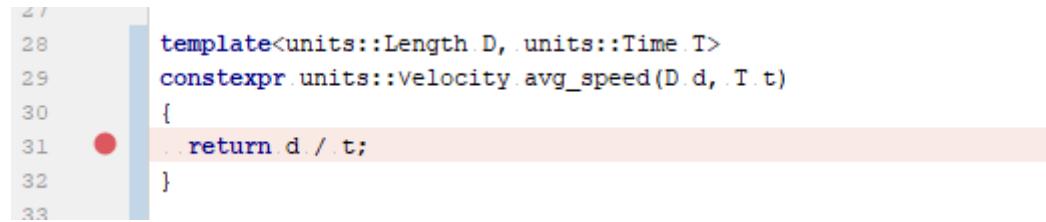
## GCC-8 (FULL ERROR LOG - CONTINUED)

```
...
include/meta/meta_fwd.hpp:206:18: note: within ‘template<class T, class U> concept const bool meta::Same<T, U>
[with T = units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, 1> >;
U = units::dimension_velocity]’
META_CONCEPT Same =
    ^~~~
include/meta/meta_fwd.hpp:206:18: note: ‘meta::detail::bool_’ evaluated to false
include/meta/meta_fwd.hpp:206:18: note: within ‘template<class T, class U> concept const bool meta::Same<T, U>
[with T = units::dimension_velocity;
U = units::dimension<units::exp<units::base_dim_length, 1>, units::exp<units::base_dim_time, 1> >]’
include/meta/meta_fwd.hpp:206:18: note: ‘meta::detail::bool_’ evaluated to false
```

Probably it will be improved even more with time...

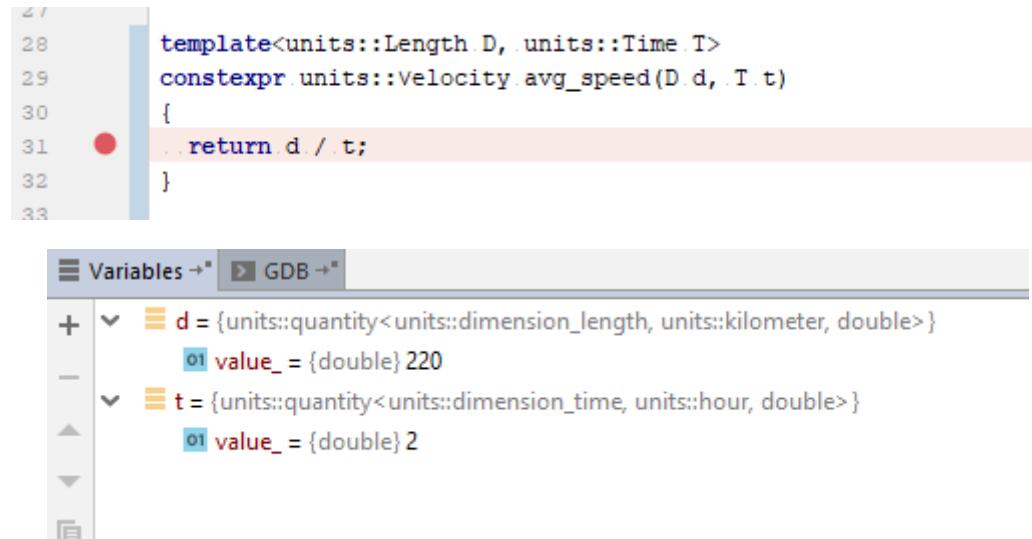
# User experience: Debugging

---



```
27
28     template<units::Length D, units::Time T>
29     constexpr units::Velocity avg_speed(D d, T t)
30     {
31         ● return d / t;
32     }
33 
```

# User experience: Debugging



The screenshot shows a debugger interface with two main panes. The top pane displays a portion of C++ code:

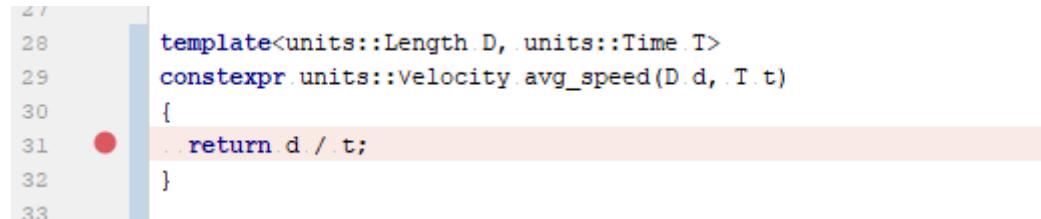
```
27
28     template<units::Length D, units::Time T>
29     constexpr units::Velocity avg_speed(D d, T t)
30     {
31         ● return d / t;
32     }
33
```

A red dot at line 31 indicates the current execution point. The bottom pane is a 'Variables' view:

Variable	Type	Value
d	{units::quantity<units::dimension_length, units::kilometer, double>}	01 value_ = {double} 220
t	{units::quantity<units::dimension_time, units::hour, double>}	01 value_ = {double} 2

# User experience: Debugging

---



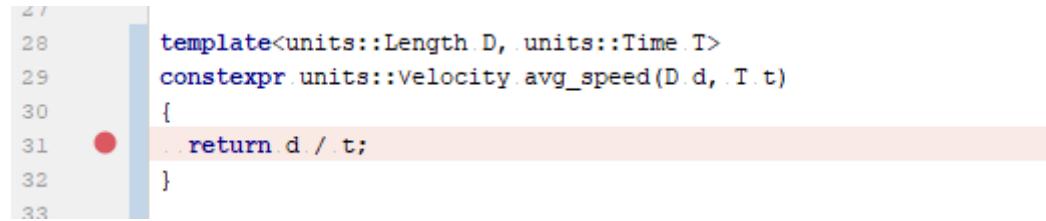
A screenshot of a debugger interface showing a code editor with a red dot at line 31, indicating a breakpoint. The code is a template function for calculating average speed.

```
27
28     template<units::Length D, units::Time T>
29     constexpr units::Velocity avg_speed(D d, T t)
30     {
31         ● return d / t;
32     }
33 
```

Breakpoint 1, (anonymous namespace)::avg\_speed<units::quantity<units::dimension\_length, units::kilometer, double>, units::quantity<units::dimension\_time, units::hour, double> > (d=..., t=...) at velocity.cpp:31  
31 return d / t;

# User experience: Debugging

---



```
27
28     template<units::Length D, units::Time T>
29     constexpr units::Velocity avg_speed(D d, T t)
30     {
31         ● return d / t;
32     }
33
```

```
(gdb) ptype d
type = class units::quantity<units::dimension_length, units::kilometer, double>
[with D = units::dimension_length, U = units::kilometer, Rep = double] {
...
}
```

# Open design questions

---

- 1 What to do with `std::chrono::duration`?
- 2 What is the best way to add support for temperatures?
- 3 Should we provide strong types and `upcasting_traits` for `quantity` types?
- 4 Should we provide aliases for quantities of units (i.e. `meters<int>`) to workaround CTAD problem?
- 5 Do we need non-linear scale? How to support it?

# Open design questions

---

- 1 What to do with `std::chrono::duration`?
- 2 What is the best way to add support for temperatures?
- 3 Should we provide strong types and `upcasting_traits` for `quantity` types?
- 4 Should we provide aliases for quantities of units (i.e. `meters<int>`) to workaround CTAD problem?
- 5 Do we need non-linear scale? How to support it?

More design questions on the project website  
(<https://github.com/mpusz/units>)

# Let's join forces!

---

We really need physical units and dimensional analysis support in the C++ Standard Library

# Let's join forces!

---

We really need physical units and dimensional analysis support in the C++ Standard Library

## WHY TO JOIN?

- C++ community and industry really need it
- Great opportunity to learn C++20
- An interesting and hard challenge to solve ;-)

# Let's join forces!

---

We really need physical units and dimensional analysis support in the C++ Standard Library

## WHY TO JOIN?

- C++ community and industry really need it
- Great opportunity to learn C++20
- An interesting and hard challenge to solve ;-)

Please, help...



**CAUTION**  
**Programming**  
**is addictive**  
**(and too much fun)**