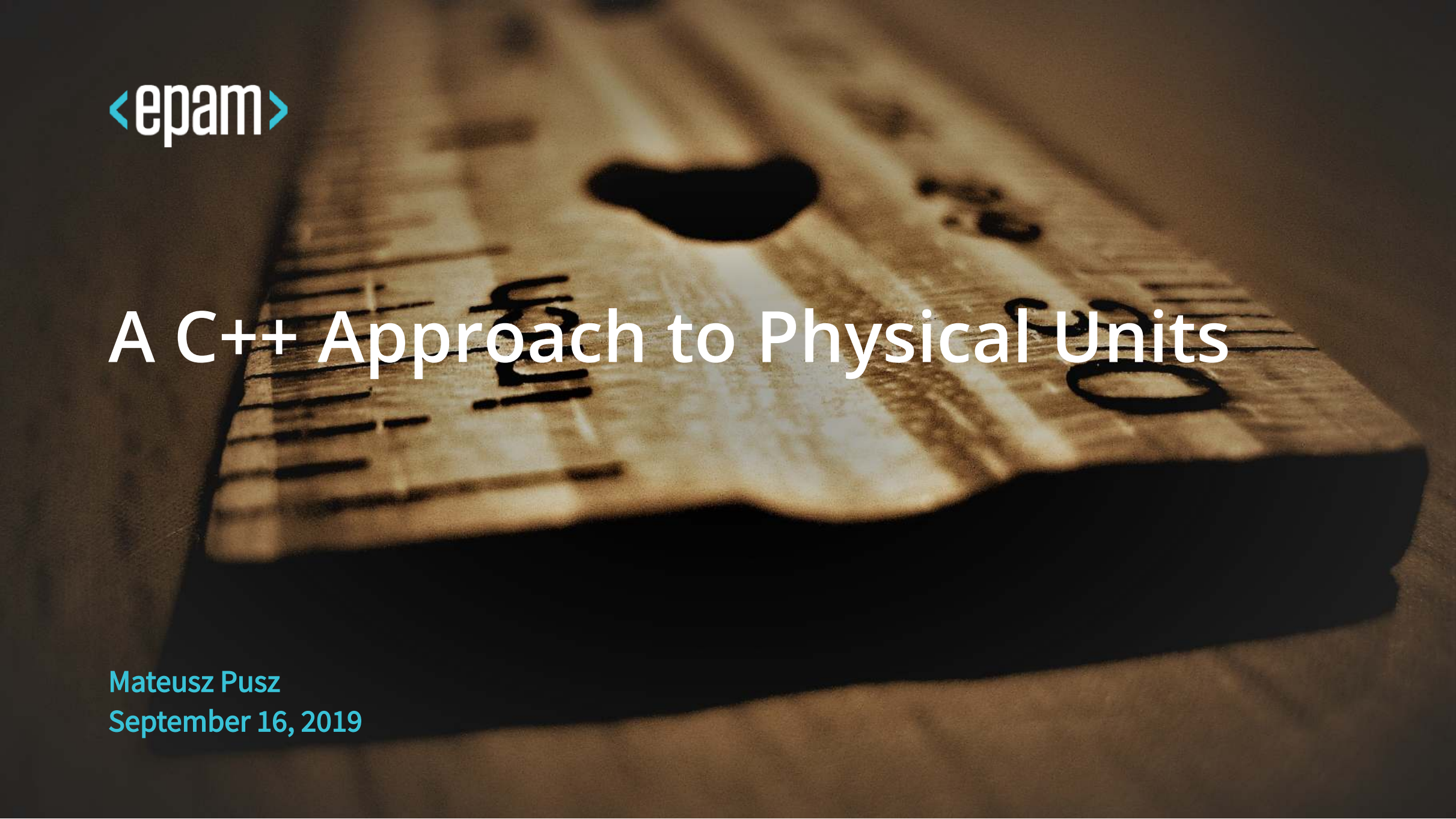


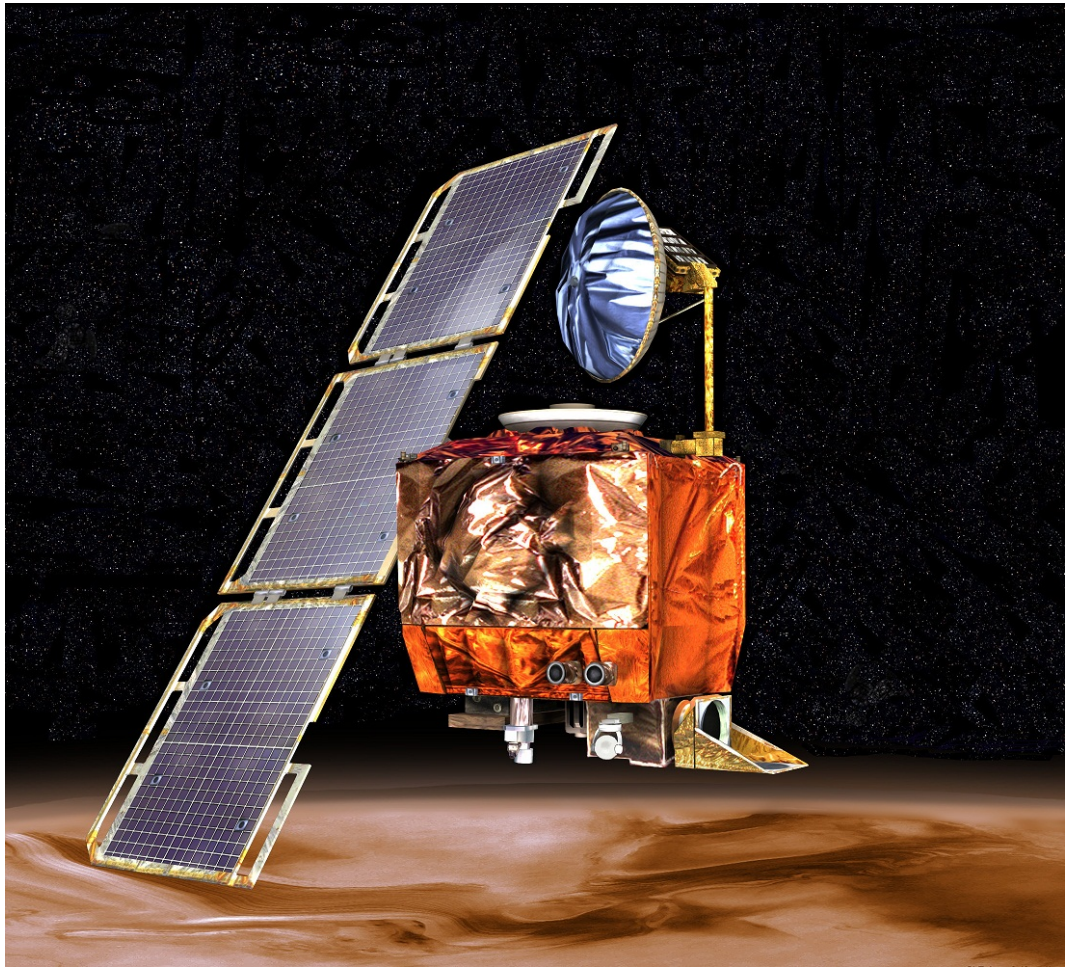
A C++ Approach to Physical Units

Mateusz Pusz
September 16, 2019

A famous motivating example

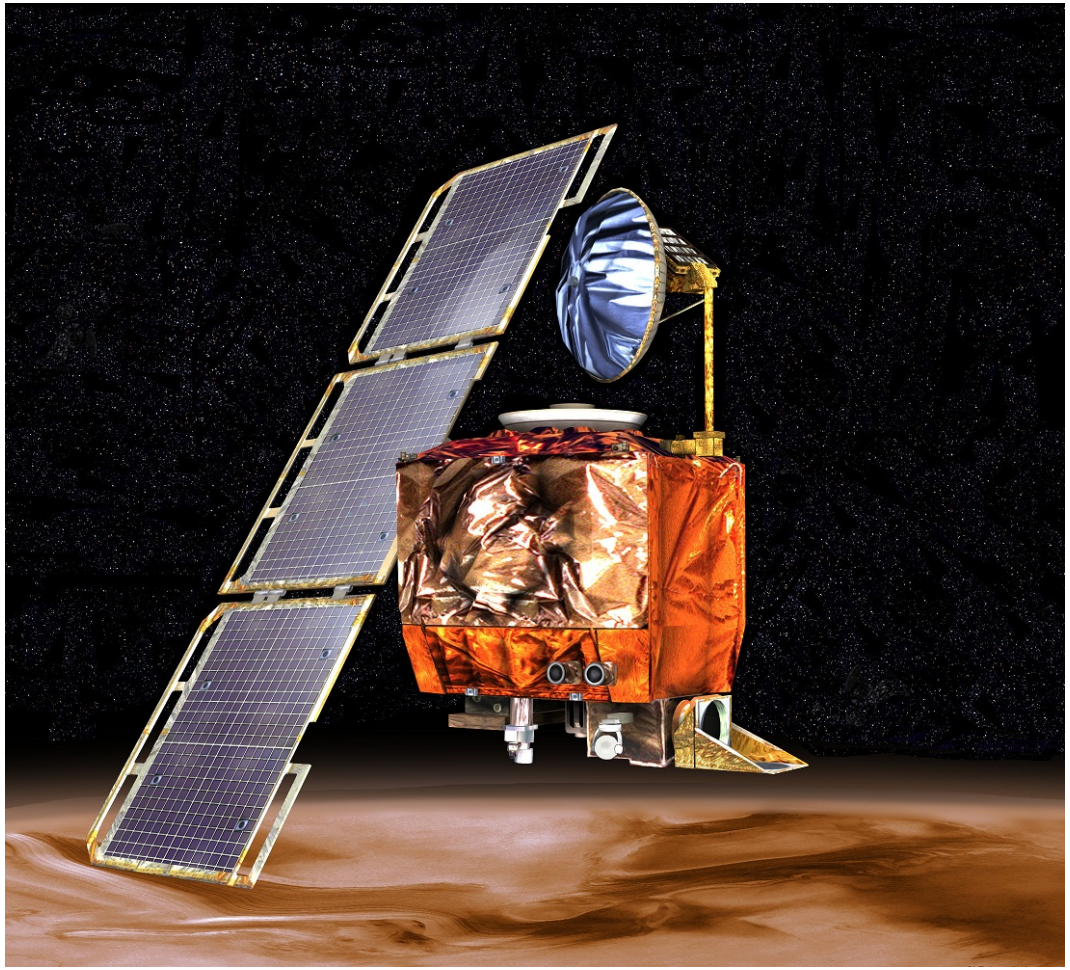
The Mars Climate Orbiter

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



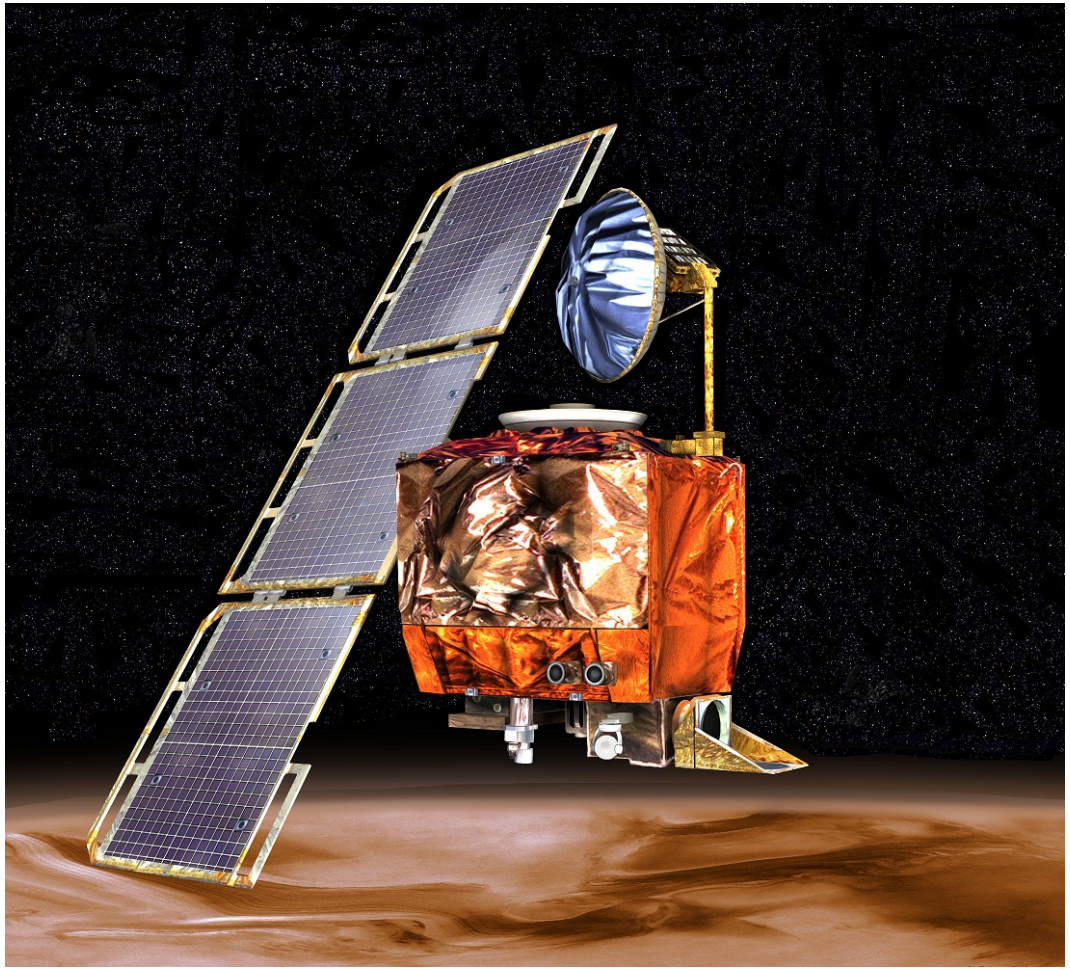
The Mars Climate Orbiter

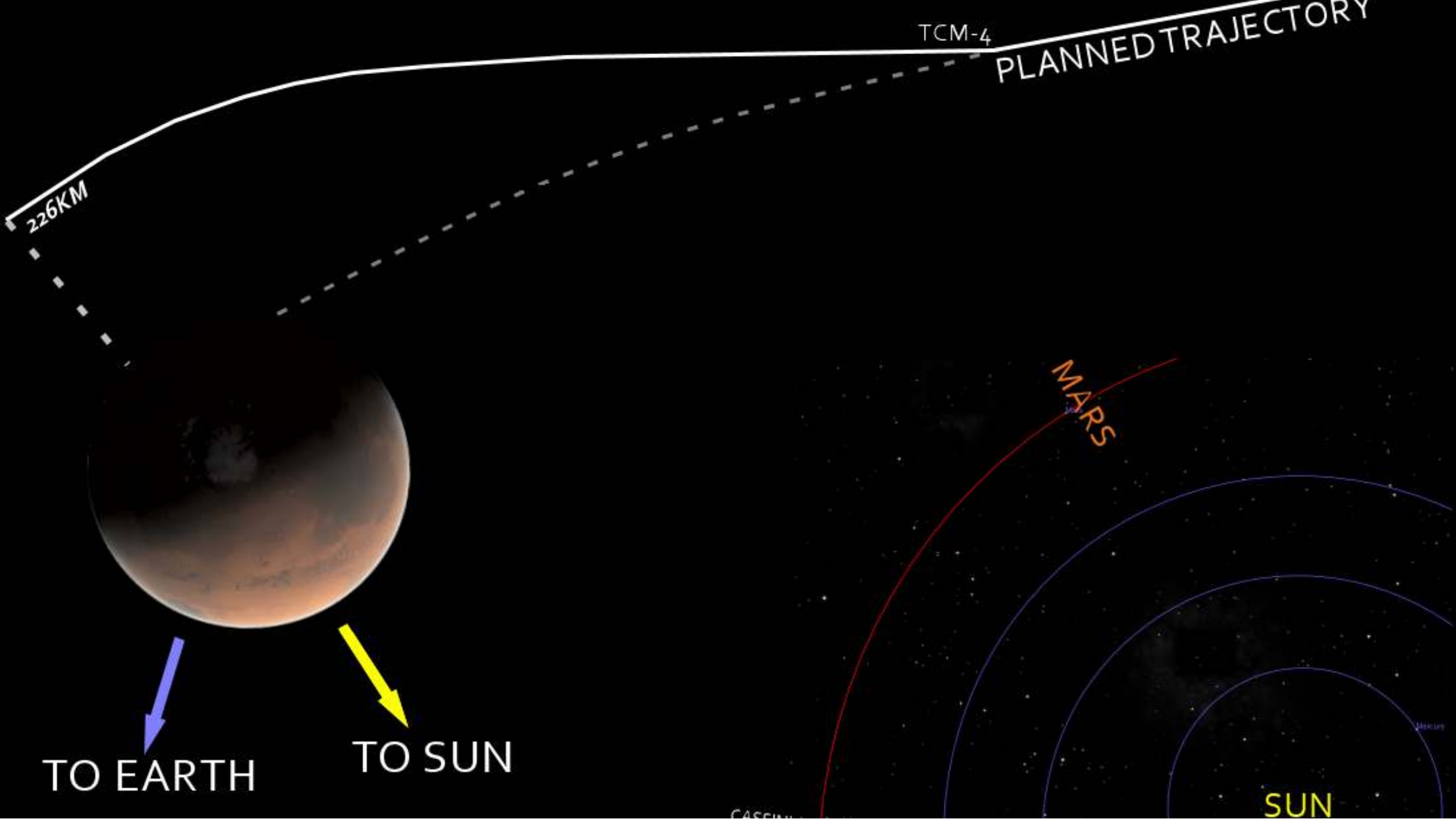
- **Robotic space probe** launched by NASA on December 11, 1998
- Project costs: **$327.6 million**
  - spacecraft development: $193.1 million
  - launching it: $91.7 million
  - mission operations: $42.8 million

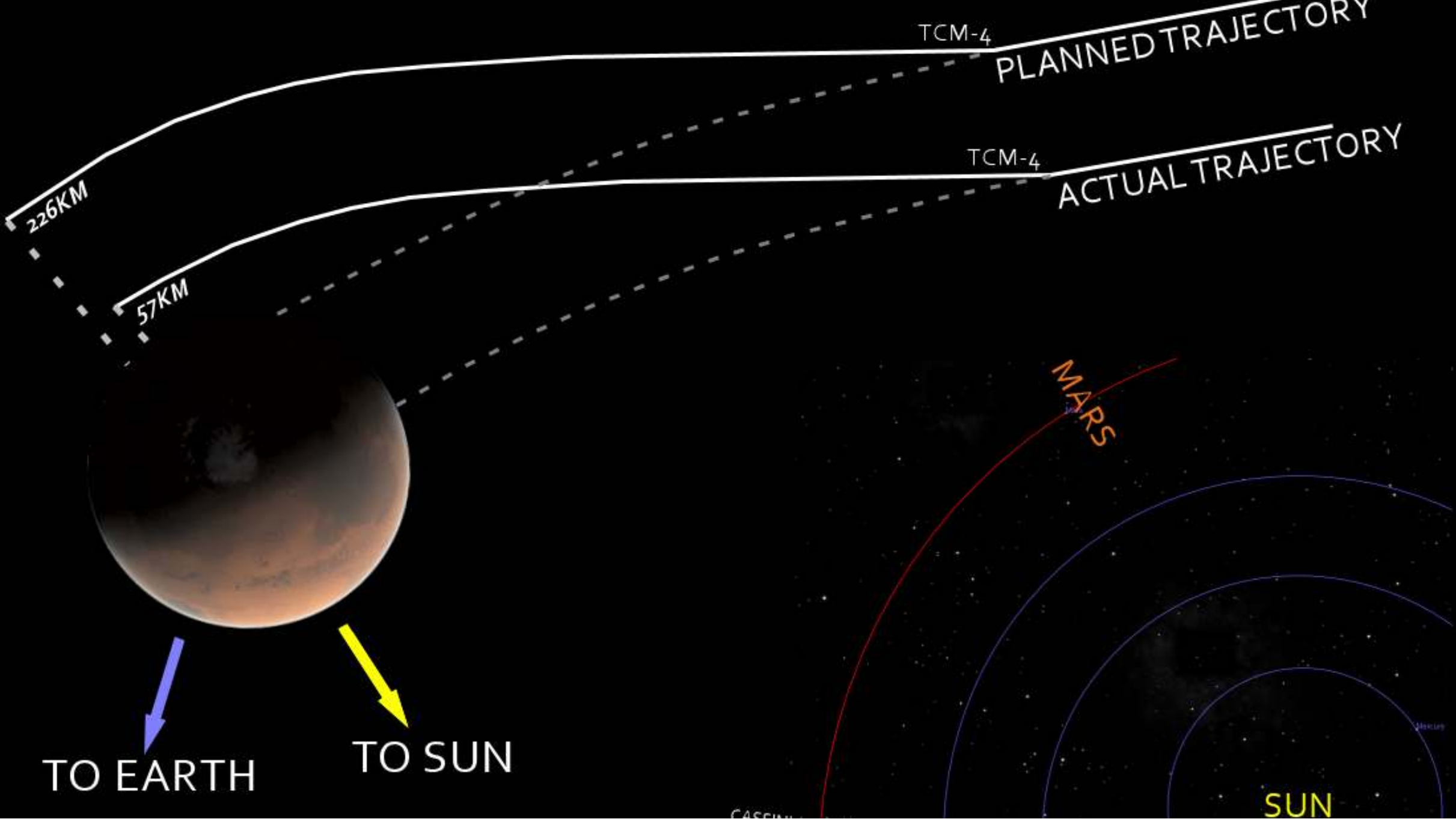


The Mars Climate Orbiter

- **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




TCM-4
PLANNED TRAJECTORY
226KM
MARS
TO EARTH
TO SUN
SUN


TCM-4
PLANNED TRAJECTORY
TCM-4
ACTUAL TRAJECTORY
226KM
57KM
MARS
TO EARTH
TO SUN
SUN

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

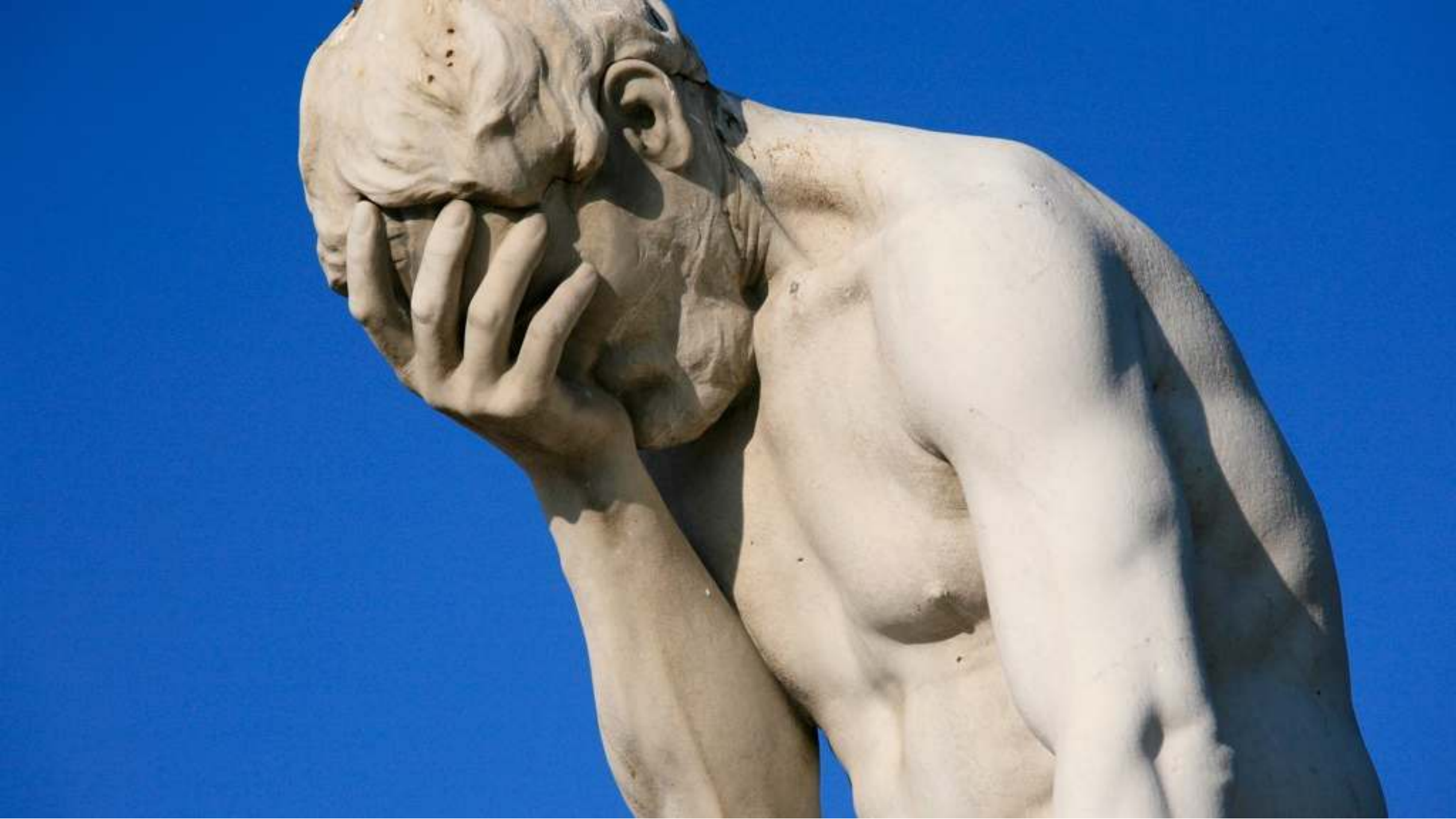
What went wrong?

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**



ANOTHER EXAMPLE

Why do I care?

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 this 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 the production code

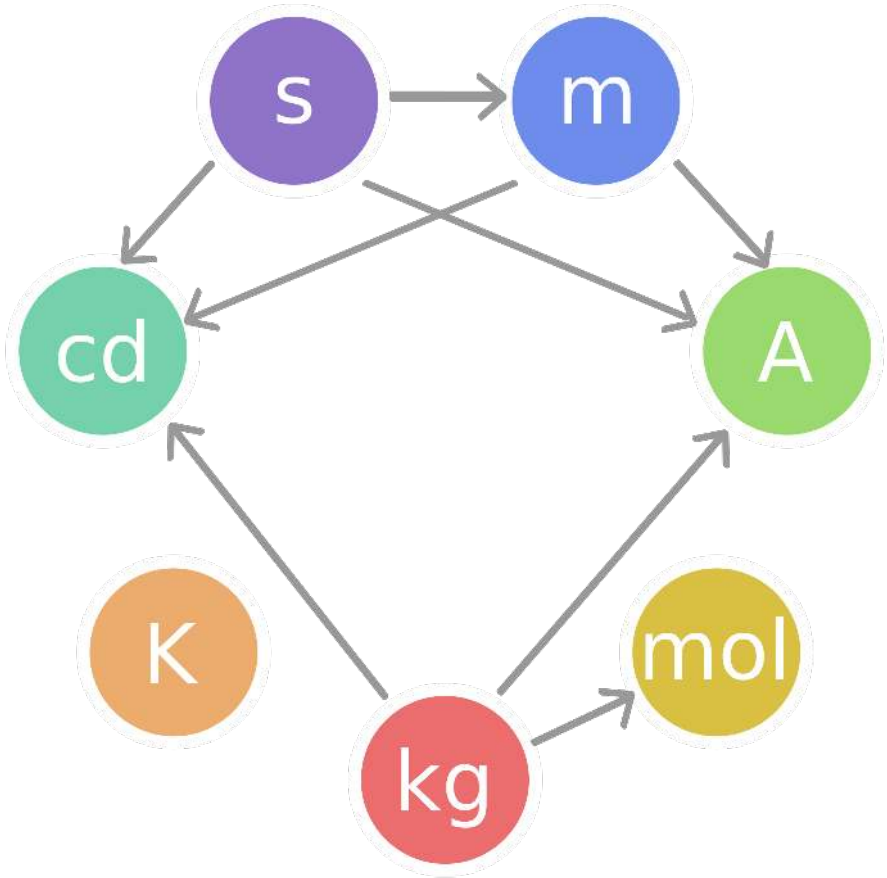
Why do we write our code this way?

- 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 the production code
- Implementing a good library by ourselves is hard

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	m^2	square metre
volume	V	m^3	cubic metre
velocity	v	$\mathrm{m \cdot s^{-1}}$	metre per second
acceleration	a	$\mathrm{m \cdot s^{-2}}$	metre per second squared
density	ρ	$\mathrm{kg \cdot m^{-3}}$	kilogram per cubic metre
magnetic field strength	H	$\mathrm{A \cdot m^{-1}}$	ampere per metre
luminance	Lv	$\mathrm{cd \cdot m^{-2}}$	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	---	s^{-1}
force	N	newton	---	$kg \cdot m \cdot s^{-2}$
pressure	Pa	pascal	N/m^2	$kg \cdot m^{-1} \cdot s^{-2}$
energy	J	joule	$N \cdot m$	$kg \cdot m^2 \cdot s^{-2}$
power	W	watt	J/s	$kg \cdot m^2 \cdot s^{-3}$
electric charge	C	coulomb	---	$s \cdot A$
voltage	V	volt	W/A	$kg \cdot m^2 \cdot s^{-3} \cdot A^{-1}$
capacitance	F	farad	C/V	$kg^{-1} \cdot m^{-2} \cdot s^4 \cdot A^2$
resistance	Ω	ohm	V/A	$kg \cdot m^2 \cdot s^{-3} \cdot A^{-2}$

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
- $N \cdot m/s$
- $kg \cdot m \cdot s^{-2} \cdot m/s$
- $kg \cdot m^2 \cdot s^{-2}/s$
- $kg \cdot m^2 \cdot s^{-3}$

Dimensional Analysis

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

Dimensional Analysis

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

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

Dimensional Analysis

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

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

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

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

static_assert(10km / 5km == 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,                  10000000000000000000> atto;
typedef ratio<1,                     1000000000000000> femto;
typedef ratio<1,                        1000000000000> pico;
typedef ratio<1,                           1000000000> nano;
typedef ratio<1,                              1000000> micro;
typedef ratio<1,                                 1000> milli;
typedef ratio<1,                                  100> centi;
typedef ratio<1,                                   10> 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<                   10000000000000000000, 1> exa;
```

More than one system of measurement

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

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

CURRENT STATE

A QUICK 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
  - no additional intermediate conversions
  - as fast as a custom code implemented with **`double`**s

Existing solutions

- **Boost.Units**
  - authors: Steven Watanabe, Matthias C. Schabel
  - https://www.boost.org/doc/libs/1_69_0/doc/html/boost_units.html

Existing solutions

- **Boost.Units**
  - authors: Steven Watanabe, Matthias C. Schabel
  - https://www.boost.org/doc/libs/1_69_0/doc/html/boost_units.html
- **NHolthaus Units**
  - author: Nic Holthaus
  - https://github.com/nholthaus/units

Existing solutions

- **Boost.Units**
  - authors: Steven Watanabe, Matthias C. Schabel
  - https://www.boost.org/doc/libs/1_69_0/doc/html/boost_units.html
- **NHolthaus 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(bu::quantity<bu::si::length>(220 * bu::si::kilo * bu::si::meters),
                         bu::quantity<bu::si::time>(2 * hours));
using kilometers_per_hour = bu::divide_typeof_helper<length_kilometer, time_hour>::type;
const 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(bu::quantity<bu::si::length>(220 * bu::si::kilo * bu::si::meters),
                         bu::quantity<bu::si::time>(2 * hours));
using kilometers_per_hour = bu::divide_typeof_helper<length_kilometer, time_hour>::type;
const bu::quantity<kilometers_per_hour> kmph(v);
std::cout << kmph.value() << " km/h\n";
```

```
const auto v = avg_speed(bu::quantity<bu::si::length>(140 * miles),
                         bu::quantity<bu::si::time>(2 * hours));
using miles_per_hour = bu::divide_typeof_helper<length_mile, time_hour>::type;
const 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(bu::quantity<bu::si::length>(220 * bu::si::kilo * bu::si::meters),
                         bu::quantity<bu::si::time>(2 * hours));
using kilometers_per_hour = bu::divide_typeof_helper<length_kilometer, time_hour>::type;
const bu::quantity<kilometers_per_hour> kmph(v);
std::cout << kmph.value() << " km/h\n";
```

```
const auto v = avg_speed(bu::quantity<bu::si::length>(140 * miles),
                         bu::quantity<bu::si::time>(2 * hours));
using miles_per_hour = bu::divide_typeof_helper<length_mile, time_hour>::type;
const bu::quantity<miles_per_hour> mph(v);
std::cout << mph.value() << " mph\n";
```

Works, but runtime does unnecessary operations, and we may loose 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; }
```

- Does it really return 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; }
```

- Does it really return 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);
```

```
const bu::quantity<length_kilometer> d = 220 * bu::si::kilo * bu::si::meters;
const 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);
```

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

```
const bu::quantity<length_mile> d = 140 * miles;
const 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)

NHolthaus Units: Toy example

```
#include "units.h"

using namespace units::literals;
```

NHolthaus 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;
}
```

NHolthaus 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;
}
```

NHolthaus 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>>>;
```

NHolthaus Units: Toy example

```
#include "units.h"

using namespace units::literals;
```

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

NHolthaus Units: Toy example

```
#include "units.h"

using namespace units::literals;
```

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

Using SFINAE in every single function template working with units is probably too complicated or time consuming for an average user of the library

NHolthaus Units: Toy example

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

NHolthaus 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";
```

NHolthaus 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";
```

NHolthaus Units: Toy example

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

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

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

NHolthaus Units: Toy example

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

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

```
const units::length::mile_t d(140);
const 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

NHolthaus Units: Summary

PROS

- *Single header* file **`units.h`**
- The conversions between units are defined as

  ratios at compile time
- *UDL support*

NHolthaus 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::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> > > > > > > > >,[...]>,[...]>’ 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

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

Copy-initialization is less permissive than **direct-initialization**: explicit constructors are not converting constructors and are not considered for copy-initialization

User experience: Compilation: Boost.Units

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

Copy-initialization is less permissive than **direct-initialization**: explicit constructors are not converting constructors and are not considered for copy-initialization

Boost.Units disallows implicit conversions even for non-narrowing operations (**km** -> **m**)

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 a shorter error message is not necessarily better ;-)

User experience: Compilation: NHolthaus Units

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

User experience: Compilation: NHolthaus Units

```cpp
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: NHolthaus 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` is often not the best solution

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

User experience: Compilation: NHolthaus Units

```cpp
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<0, 1>, ratio<0, 1>, ratio<0, 1>, ratio<0, 1>, ratio<0, 1> >, ratio<0, 1>,
ratio<0, 1> >, unit<ratio<5, 18>, base_unit<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> >, ratio<0, 1>, ratio<0, 1> > >::value' "Units are not compatible."
                static_assert(traits::is_convertible_unit<UnitFrom, UnitTo>::value, "Units are not compatible.");
                ^             ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

A need to modernize our toolbox

- For most template metaprogramming libraries *compile-time errors are rare*

A need to modernize our toolbox

- For most template metaprogramming libraries *compile-time errors are rare*
- It is expected that engineers working with a physical units library **will experience compile-time errors very often**
  - generating compile-time errors for invalid calculation is the *main reason to create such a library*

A need to modernize our toolbox

- For most template metaprogramming libraries *compile-time errors are rare*
- It is expected that engineers working with a physical units library **will experience compile-time errors very often**
  - generating compile-time errors for invalid calculation is the *main reason to create such a library*

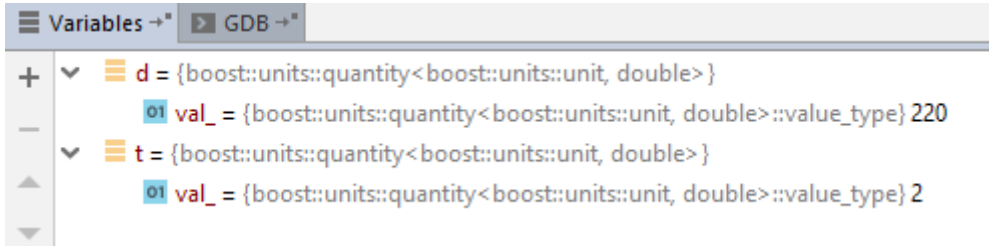
In case of the physical units library we have to rethink the way we do template metaprogramming!

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
```

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
```



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
```

```
Breakpoint 1, 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
```

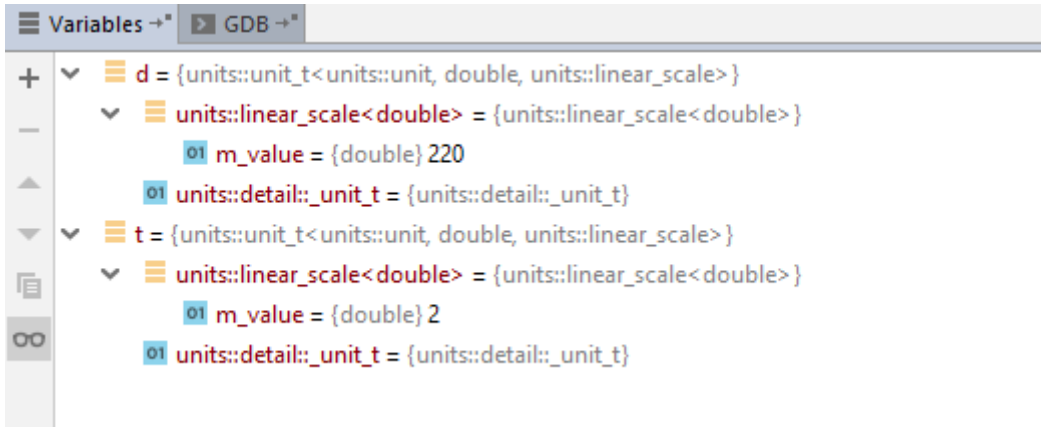
```
(gdb) ptype d
type = class boost::units::quantity<boost::units::unit<boost::units::list<boost::units::dim<boost::units::length_base_dimension,
boost::units::static_rational<1, 1> >, boost::units::dimensionless_type>, boost::units::heterogeneous_system
<boost::units::heterogeneous_system_impl<boost::units::list<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: NHolthaus 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  }
```

User experience: Debugging: NHolthaus 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  }
```



User experience: Debugging: NHolthaus 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  }
```

```
Breakpoint 1, 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: NHolthaus 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: NHolthaus 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, astronicalUnit, astronicalUnits, 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>, astronicalUnits, 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

NHOLTHAUS 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

Requirements

- The best possible user experience
  - compiler errors
  - debugging

Requirements

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

Requirements

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

Requirements

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

A toy example

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

A toy example

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

```
const Velocity auto speed = avg_speed(220.km, 2.h);
assert(speed.count() == 110);
std::cout << quantity_cast<kilometre_per_hour>(speed).count() << " km/h\n";
```

A toy example

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

```
const Velocity auto speed = avg_speed(220.km, 2.h);
assert(speed.count() == 110);
std::cout << quantity_cast<kilometre_per_hour>(speed).count() << " km/h\n";
```

```
const Velocity auto speed = avg_speed(140.mi, 2.h);
assert(speed.count() == 70);
std::cout << quantity_cast<mile_per_hour>(speed).count() << " mph\n";
```

A toy example

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

```
const Velocity auto speed = avg_speed(220.km, 2.h);
assert(speed.count() == 110);
std::cout << quantity_cast<kilometre_per_hour>(speed).count() << " km/h\n";
```

```
const Velocity auto speed = avg_speed(140.mi, 2.h);
assert(speed.count() == 70);
std::cout << quantity_cast<mile_per_hour>(speed).count() << " mph\n";
```

No intermediate conversion to SI base units and back

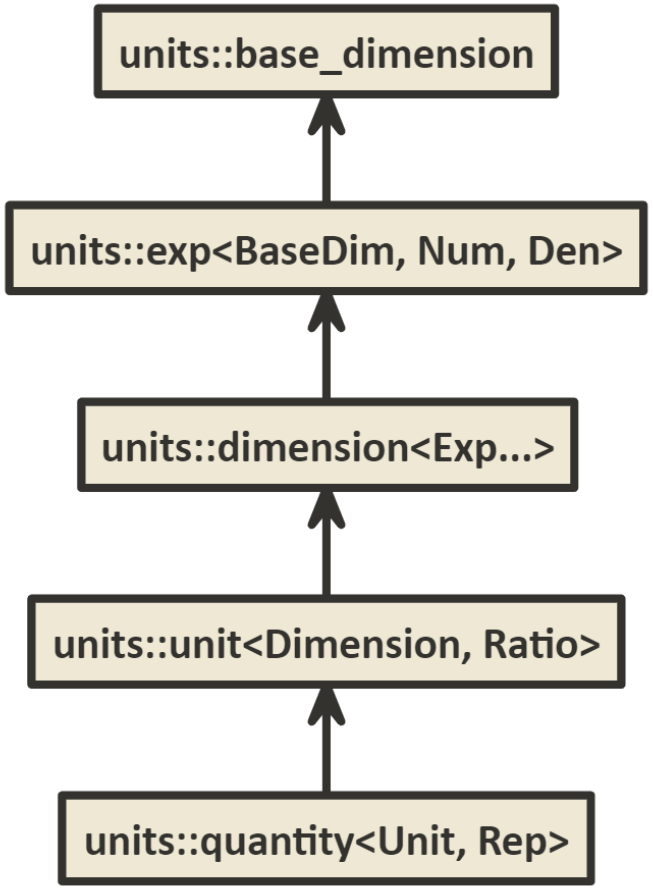
A toy example

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

```
const quantity<kilometre> d(220);
const quantity<hour> t(2);
const Velocity auto kmph = quantity_cast<kilometre_per_hour>(avg_speed(d, t));
std::cout << kmph.count() << " km/h\n";
```

```
const quantity<mile> d(140);
const quantity<hour> t(2);
const Velocity auto mph = quantity_cast<mile_per_hour>(avg_speed(d, t));
std::cout << mph.count() << " mph\n";
```

Design overview



Base Dimensions

- **`units::base_dimension`** - a *unique sortable compile-time* identifier of a base dimension

```
struct base_dimension {
  const char* name;
};
constexpr bool operator==(const base_dimension& lhs, const base_dimension& rhs);
constexpr bool operator<(const base_dimension& lhs, const base_dimension& rhs);
```

Base Dimensions

- **`units::base_dimension`** - a *unique sortable compile-time* identifier of a base dimension

```
struct base_dimension {
  const char* name;
};
constexpr bool operator==(const base_dimension& lhs, const base_dimension& rhs);
constexpr bool operator<(const base_dimension& lhs, const base_dimension& rhs);
```

EXAMPLE

```
inline constexpr base_dimension base_dim_length{"length"};
inline constexpr base_dimension base_dim_mass{"mass"};
```

Base Dimensions

- **`units::base_dimension`** - a *unique sortable compile-time* identifier of a base dimension

```
struct base_dimension {
  const char* name;
};
constexpr bool operator==(const base_dimension& lhs, const base_dimension& rhs);
constexpr bool operator<(const base_dimension& lhs, const base_dimension& rhs);
```

EXAMPLE

```
inline constexpr base_dimension base_dim_length{"length"};
inline constexpr base_dimension base_dim_mass{"mass"};
```

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

Derived Dimensions

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

```
template<const base_dimension& BaseDimension, int Num, int Den = 1>
struct exp {
  static constexpr const base_dimension& dimension = BaseDimension;
  static constexpr int num = Num;
  static constexpr int den = Den;
};
```

Derived Dimensions

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

```
template<Exponent... Es>
struct dimension;
```

Derived Dimensions

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

```
template<Exponent... Es>
struct dimension;
```

EXAMPLE

```
using my_velocity = 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 with a type-list approach

- The *same type expected* for both operations

```
constexpr auto v1 = 1m / 1s;
constexpr auto v2 = 2 / 2s * 1m;

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

`make_dimension` factory helper

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

template<Exponent... Es>
using make_dimension_t = make_dimension<Es...>::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*

make_dimension factory helper

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

template<Exponent... Es>
using make_dimension_t = make_dimension<Es...>::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

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

Units

- **`units::unit`** - the unit of a specific physical dimension

```
template<Dimension D, Ratio R = ratio<1>>
  requires (R::num * R::den > 0)
struct unit {
  using dimension = D;
  using ratio = R;
};
```

Units

- **`units::unit`** - the unit of a specific physical dimension

```
template<Dimension D, Ratio R = ratio<1>>
  requires (R::num * R::den > 0)
struct unit {
  using dimension = D;
  using ratio = R;
};
```

EXAMPLE

```
struct metre : unit<length> {};
```

Units

- **`units::unit`** - the unit of a specific physical dimension

```
template<Dimension D, Ratio R = ratio<1>>
  requires (R::num * R::den > 0)
struct unit {
  using dimension = D;
  using ratio = R;
};
```

EXAMPLE

```
struct metre : unit<length> {};
```

```
struct kilometre : kilo<metre> {};
```

Units

- **`units::unit`** - the unit of a specific physical dimension

```
template<Dimension D, Ratio R = ratio<1>>
  requires (R::num * R::den > 0)
struct unit {
  using dimension = D;
  using ratio = R;
};
```

EXAMPLE

```
struct metre : unit<length> {};
```

```
struct kilometre : kilo<metre> {};
```

```
struct kilometre_per_hour : derived_unit<velocity, kilometre, hour> {};
```

Quantities

- **`units::quantity`** - the *amount of a specific dimension* expressed *in a specific unit* of this dimension

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

Quantities

- **`units::quantity`** - the *amount of a specific dimension* expressed *in a specific unit* of this dimension

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

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

```
1kmph * 1h == 1km
```

  - division of 2 quantities of different dimensions

```
1km / 1h == 1kmph
```

  - division of a scalar with a quantity

```
1 / 1s == 1Hz
```

Quantities

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

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

- Interface similar to ***`std::chrono::duration`*** *+ additional member functions*
  - **`std::common_type_t`** removed from the return type of most of the operators

```
const auto r1 = v1 * v2;
const auto r2 = v1 / v2;

static_assert(std::is_same<decltype(r1)>, std::common_type_t<r1, r2>); // fails for some types
static_assert(std::is_same<decltype(r2)>, std::common_type_t<r1, r2>); // fails for some types
```

  - quantity value equal to **`1`** expressed in a specific **`Rep`**

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

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process
- As a result **end users** get **huge types in error messages**

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process
- As a result **end users** get **huge types in error messages**

EXAMPLE

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
using kilometre_per_hour = derived_unit<velocity, kilometre, hour>;
```

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process
- As a result **end users** get **huge types in error messages**

EXAMPLE

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
using kilometre_per_hour = derived_unit<velocity, kilometre, hour>;
```

```
void foo(quantity<kilometre_per_hour>(90));
```

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process
- As a result **end users** get **huge types in error messages**

EXAMPLE

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
using kilometre_per_hour = derived_unit<velocity, kilometre, hour>;
```

```
void foo(quantity<kilometre_per_hour>(90));
```

```
[with Q = units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1, 1>,
   units::exp<units::base_dim_time, 1, -1> >, units::ratio<5, 18> >, double>]
```

Type aliases are great for developers but not for end users

- **Developers** cannot live without aliases as they hugely *simplify code development and its maintenance*
- Type *aliases names are lost* quickly during compilation process
- As a result **end users** get **huge types in error messages**

EXAMPLE

```
using velocity = make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>>;
using kilometre_per_hour = derived_unit<velocity, kilometre, hour>;
```

```
void foo(quantity<kilometre_per_hour>(90));
```

```
[with Q = units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1, 1>,
   units::exp<units::base_dim_time, 1, -1> >, units::ratio<5, 18> >, double>]
```

It is a pity that we still do not have **strong typedefs** in the C++ language :-(

Inheritance as a workaround

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

Inheritance as a workaround

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

- **Similarly** to strong typedefs
  - *strong types* that do not vanish during compilation process
  - member functions of a base class *taking the base class type as an argument* will still *work with a child class type* provided instead (i.e. **op==**)

Inheritance as a workaround

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

- **Similarly** to strong typedefs
  - *strong types* that do not vanish during compilation process
  - member functions of a base class *taking the base class type as an argument* will still *work with a child class type* provided instead (i.e. **op==**)
- **Alternatively** to strong typedefs
  - do not automatically inherit *constructors and assignment operators*
  - member functions of a base class *returning the base class type* will still *return the same base type for a child class instance*

Inheritance as a workaround

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

- **Similarly** to strong typedefs
  - *strong types* that do not vanish during compilation process
  - member functions of a base class *taking the base class type as an argument* will still *work with a child class type* provided instead (i.e. **op==**)
- **Alternatively** to strong typedefs
  - do not automatically inherit *constructors and assignment operators*
  - member functions of a base class *returning the base class type* will still *return the same base type for a child class instance*

Easy to apply for **`dimension`** and **`unit`**, much harder for a **`quantity`** type

Type substitution problem

```
Velocity auto v = 10m / 2s;
```

Type substitution problem

```
Velocity auto v = 10m / 2s;
```

```
template<typename U1, typename Rep1, typename U2, typename Rep2>
[[nodiscard]] constexpr Quantity operator/(const quantity<U1, Rep1>& lhs,
                                           const quantity<U2, Rep2>& rhs);
```

Type substitution problem

```
Velocity auto v = 10m / 2s;
```

```
template<typename U1, typename Rep1, typename U2, typename Rep2>
[[nodiscard]] constexpr Quantity operator/(const quantity<U1, Rep1>& lhs,
                                           const quantity<U2, Rep2>& rhs);
```

How to form a **`velocity`** dimension child class from division of **`length`** by **`time`**?

Downcasting facility

BASE CLASS

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

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

Downcasting facility

BASE CLASS

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

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

DOWNCASTABLE

```
template<typename T>
concept Downcastable =
    requires {
      typename T::base_type;
    } &&
    std::derived_from<T, downcast_base<typename T::base_type>>;
```

Downcasting facility

HELPER ALIASES

```
template<Downcastable T>
using downcast_from = T::base_type;

template<Downcastable T>
using downcast_to = std::type_identity<T>;
```

Downcasting facility

HELPER ALIASES

```
template<Downcastable T>
using downcast_from = T::base_type;

template<Downcastable T>
using downcast_to = std::type_identity<T>;
```

DOWNCASTING TRAITS

```
template<Downcastable T>
struct downcasting_traits : downcast_to<T> {};

template<Downcastable T>
using downcasting_traits_t = downcasting_traits<T>::type;
```

Downcasting `dimension`

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

Downcasting `dimension`

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

```
struct velocity : make_dimension_t<exp<base_dim_length, 1>, exp<base_dim_time, -1>> {};
template<>
struct downcasting_traits<downcast_from<velocity>> : downcast_to<velocity> {};
```

Downcasting `unit`

```
template<Dimension D, Ratio R = ratio<1>>
  requires (R::num * R::den > 0)
struct unit : downcast_base<unit<D, R>> {
  using dimension = D;
  using ratio = R;
};
```

Downcasting `unit`

```
template<Dimension D, Ratio R = ratio<1>>
  requires (R::num * R::den > 0)
struct unit : downcast_base<unit<D, R>> {
  using dimension = D;
  using ratio = R;
};
```

```
struct metre_per_second : derived_unit<velocity, metre, second> {};
template<>
struct downcasting_traits<downcast_from<metre_per_second>> : downcast_to<metre_per_second> {};
```

Downcasting facility

BEFORE

```
[with D = units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1, 1>,
                  units::exp<units::base_dim_time, 1, -1> >, units::ratio<5, 18> >, double>]
```

Downcasting facility

BEFORE

```
[with D = units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1, 1>,
                units::exp<units::base_dim_time, 1, -1> >, units::ratio<5, 18> >, double>]
```

AFTER

```
[with D = units::quantity<units::kilometre_per_hour, double>]
```

Conceptify all the things

All template types are heavily embraced with concepts

Conceptify all the things

All template types are heavily embraced with concepts

UNITS ENGINE CONCEPTS

- `TypeList`
- `Scalar`
- `Ratio`
- `Exponent`
- `Dimension`
- `Unit`
- `Quantity`

Conceptify all the things

All template types are heavily embraced with concepts

UNITS ENGINE CONCEPTS

- `TypeList`
- `Scalar`
- `Ratio`
- `Exponent`
- `Dimension`
- `Unit`
- `Quantity`

PREDEFINED QUANTITIES CONCEPTS

- `Length`
- `Time`
- `Frequency`
- `Velocity`
- …

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

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;
}
```

```
example.cpp: In instantiation of ‘constexpr units::Velocity avg_speed(D, T)
    [with D = units::quantity<units::kilometre>; T = units::quantity<units::hour>]’:
example.cpp:49:49:   required from here
example.cpp:34:14: error: placeholder constraints not satisfied
    34 |   return d * t;
       |              ^
include/units/dimensions/velocity.h:34:16: note: within ‘template<class T> concept units::Velocity<T>
    [with T = units::quantity<units::unit<units::dimension<units::exp<units::base_dim_length, 1, 1>,
                units::exp<units::base_dim_time, 1, 1> >, units::ratio<3600000, 1> >, double>]’
    34 |   concept Velocity = Quantity<T> && std::same_as<typename T::dimension, velocity>;
       |           ^~~~~~~~
include/stl2/detail/concepts/core.hpp:37:15: note: within ‘template<class T, class U> concept std::same_as<T, U>
    [with T = units::dimension<units::exp<units::base_dim_length, 1, 1>, units::exp<units::base_dim_time, 1, 1> >;
          U = units::velocity]’
    37 |  META_CONCEPT same_as = 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;
}
```

```
include/meta/meta_fwd.hpp:224:18: note: within ‘template<class T, class U> concept meta::Same<T, U>
    [with T = units::dimension<units::exp<units::base_dim_length, 1, 1>, units::exp<units::base_dim_time, 1, 1> >;
          U = units::velocity]’
   224 |     META_CONCEPT Same =
       |                  ^~~~
include/meta/meta_fwd.hpp:224:18: note: ‘meta::detail::barrier’ evaluated to false
include/meta/meta_fwd.hpp:224:18: note: within ‘template<class T, class U> concept meta::Same<T, U>
    [with T = units::velocity;
          U = units::dimension<units::exp<units::base_dim_length, 1, 1>, units::exp<units::base_dim_time, 1, 1> >]’
include/meta/meta_fwd.hpp:224:18: note: ‘meta::detail::barrier’ evaluated to false
```

User experience: Compilation

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

```
include/meta/meta_fwd.hpp:224:18: note: within ‘template<class T, class U> concept meta::Same<T, U>
    [with T = units::dimension<units::exp<units::base_dim_length, 1, 1>, units::exp<units::base_dim_time, 1, 1> >;
          U = units::velocity]’
   224 |     META_CONCEPT Same =
       |                   ^~~~
include/meta/meta_fwd.hpp:224:18: note: ‘meta::detail::barrier’ evaluated to false
include/meta/meta_fwd.hpp:224:18: note: within ‘template<class T, class U> concept meta::Same<T, U>
    [with T = units::velocity;
          U = units::dimension<units::exp<units::base_dim_length, 1, 1>, units::exp<units::base_dim_time, 1, 1> >]’
include/meta/meta_fwd.hpp:224:18: note: ‘meta::detail::barrier’ evaluated to false
```

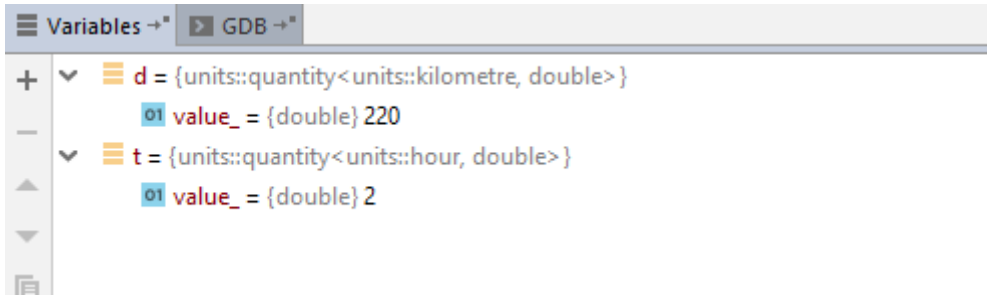
Concepts support in C++ compilers is still experimental and largely based on Concepts TS. We can expect even better diagnostics in the future.

User experience: Debugging

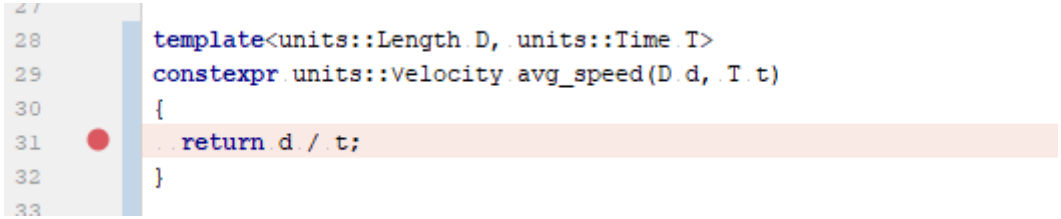
```
28  template<units::Length D, units::Time T>
29  constexpr units::Velocity avg_speed(D d, T t)
30  {
31      return d / t;
32  }
```

User experience: Debugging

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



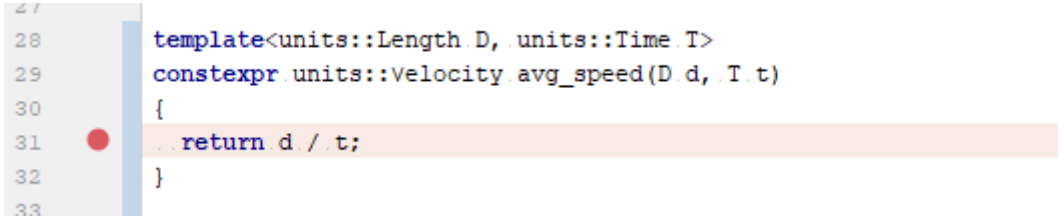
User experience: Debugging



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

```
Breakpoint 1, avg_speed<units::quantity<units::kilometre, double>,
                        units::quantity<units::hour, double> >
(d=..., t=...) at velocity.cpp:31
31      return d / t;
```

User experience: Debugging



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

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

Contracts

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

Contracts

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

```
error: macro "Expects" passed 2 arguments, but takes just 1
```

Contracts

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

Contracts

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

Contracts

```
template<Unit U, Scalar Rep = double>
class quantity {
public:
  template<typename U1, typename Rep1, typename U2, typename Rep2>
    requires std::same_as<typename U1::dimension, typename U2::dimension>
  [[nodiscard]] constexpr Scalar operator/(const quantity<U1, Rep1>& lhs,
                                           const quantity<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*

(Not C++20) Contracts

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

(Not C++20) Contracts

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

- **Contract always included in the function's signature**
  - clearly *communicates the interface between the author and the user* of the interface
  - *available* even if only a *function declaration* is available
  - *always up to date* (contrary to often outdated info in comments)

(Not C++20) Contracts

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

- **Contract always included in the function's signature**
  - clearly *communicates the interface between the author and the user* of the interface
  - *available* even if only a *function declaration* is available
  - *always up to date* (contrary to often outdated info in comments)
- **No preprocessor macros!**

Adding your own dimensions and units

DIGITAL INFORMATION

```
static_assert(1_B == 8_b);
```

Adding your own dimensions and units

DIGITAL INFORMATION

```
static_assert(1_B == 8_b);
```

BASE DIMENSION

```
inline constexpr units::base_dimension base_dim_digital_information{"digital information"};
```

Adding your own dimensions and units

DIGITAL INFORMATION

```
static_assert(1_B == 8_b);
```

BASE DIMENSION

```
inline constexpr units::base_dimension base_dim_digital_information{"digital information"};
```

DIMENSION AND ITS CONCEPT

```
struct digital_information : units::make_dimension_t<units::exp<base_dim_digital_information, 1>> {};
template<>
struct units::downcasting_traits<units::downcast_from<digital_information>> : units::downcast_to<digital_information> {};
```

Adding your own dimensions and units

DIGITAL INFORMATION

```
static_assert(1_B == 8_b);
```

BASE DIMENSION

```
inline constexpr units::base_dimension base_dim_digital_information{"digital information"};
```

DIMENSION AND ITS CONCEPT

```
struct digital_information : units::make_dimension_t<units::exp<base_dim_digital_information, 1>> {};
template<>
struct units::downcasting_traits<units::downcast_from<digital_information>> : units::downcast_to<digital_information> {};
```

```
template<typename T>
concept DigitalInformation = units::QuantityOf<T, digital_information>;
```

Adding your own dimensions and units

DIGITAL INFORMATION

```
static_assert(1_B == 8_b);
```

UNITS

```
struct bit : units::unit<digital_information> {};
template<> struct units::downcasting_traits<units::downcast_from<bit>> : units::downcast_to<bit> {};

struct byte : units::unit<digital_information, units::ratio<8>> {};
template<> struct units::downcasting_traits<units::downcast_from<byte>> : units::downcast_to<byte> {};
```

Adding your own dimensions and units

DIGITAL INFORMATION

```
static_assert(1_B == 8_b);
```

UNITS

```
struct bit : units::unit<digital_information> {};
template<> struct units::downcasting_traits<units::downcast_from<bit>> : units::downcast_to<bit> {};

struct byte : units::unit<digital_information, units::ratio<8>> {};
template<> struct units::downcasting_traits<units::downcast_from<byte>> : units::downcast_to<byte> {};
```

UDLS

```
inline namespace literals {
  constexpr auto operator""_b(unsigned long long l) { return units::quantity<bit, std::int64_t>(l); }
  constexpr auto operator""_b(long double l) { return units::quantity<bit, long double>(l); }

  constexpr auto operator""_B(unsigned long long l) { return units::quantity<byte, std::int64_t>(l); }
  constexpr auto operator""_B(long double l) { return units::quantity<byte, long double>(l); }
}
```

Try it yourself: https://godbolt.org/z/iEFwID (Thank You Matt!)

```
1   #include <units/quantity.h>
2
3   using namespace std::experimental;
4
5   inline constexpr units::base_dimension base_dim_digital_information{"digital information"};
6
7   struct digital_information : units::make_dimension_t<units::exp<base_dim_digital_information, 1>> {};
8   template<> struct units::downcasting_traits<units::downcast_from<digital_information>> : units::downcast_to<digital_informa
9
10  template<typename T>
11  concept DigitalInformation = units::QuantityOf<T, digital_information>;
12
13  struct bit : units::unit<digital_information> {};
14  template<> struct units::downcasting_traits<units::downcast_from<bit>> : units::downcast_to<bit> {};
15  struct byte : units::unit<digital_information, units::ratio<8>> {};
16  template<> struct units::downcasting_traits<units::downcast_from<::byte>> : units::downcast_to<::byte> {};
17
18  inline namespace literals {
19    constexpr auto operator""_b(unsigned long long l) { return units::quantity<bit, std::int64_t>(l); }
20    constexpr auto operator""_b(long double l) { return units::quantity<bit, long double>(l); }
21    constexpr auto operator""_B(unsigned long long l) { return units::quantity<byte, std::int64_t>(l); }
22    constexpr auto operator""_B(long double l) { return units::quantity<byte, long double>(l); }
23  }
24
25  static_assert(1_B == 8_b);
26
```

x86-64 gcc (trunk) | -std=c++2a -fconcepts -O2

A | 11010 | ./a.out | .LX0: | lib.f: | .text | // | \s+ | Intel | Demangle

Libraries | Add new... | Add tool...

```
1   <No assembly generated>
```

A | Wrap lines

```
Compiler returned: 0
```

Edit on Compiler Explorer

Next steps

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

Next steps

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

1. Gather *feedback and more requirements* from users

Next steps

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

1. Gather *feedback and more requirements* from users
2. First *design review on the ISO C++ Committee Meeting* in Belfast, Northern Ireland
   - possible design update according to the Committee input

Next steps

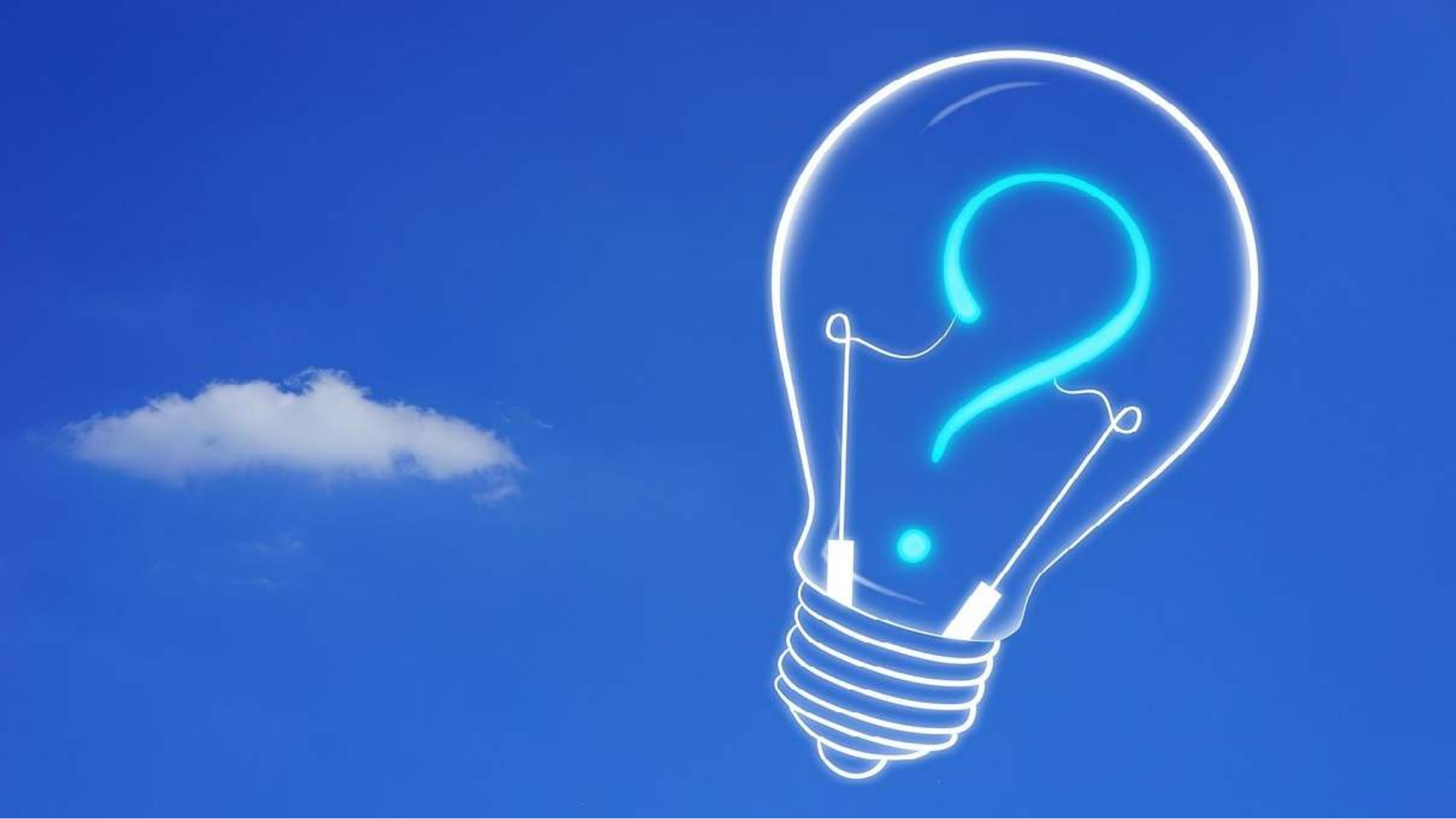
We really need physical units and dimensional analysis support in the C++ Standard Library

1. Gather *feedback and more requirements* from users
2. First *design review on the ISO C++ Committee Meeting* in Belfast, Northern Ireland
   - possible design update according to the Committee input
3. Provide *support for remaining dimensions and units*
   - based on user's feedback

Next steps

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

1. Gather *feedback and more requirements* from users
2. First *design review on the ISO C++ Committee Meeting* in Belfast, Northern Ireland
   - possible design update according to the Committee input
3. Provide *support for remaining dimensions and units*
   - based on user's feedback
4. Try to *catch C++23 train*




CAUTION
Programming
is addictive
(and too much fun)