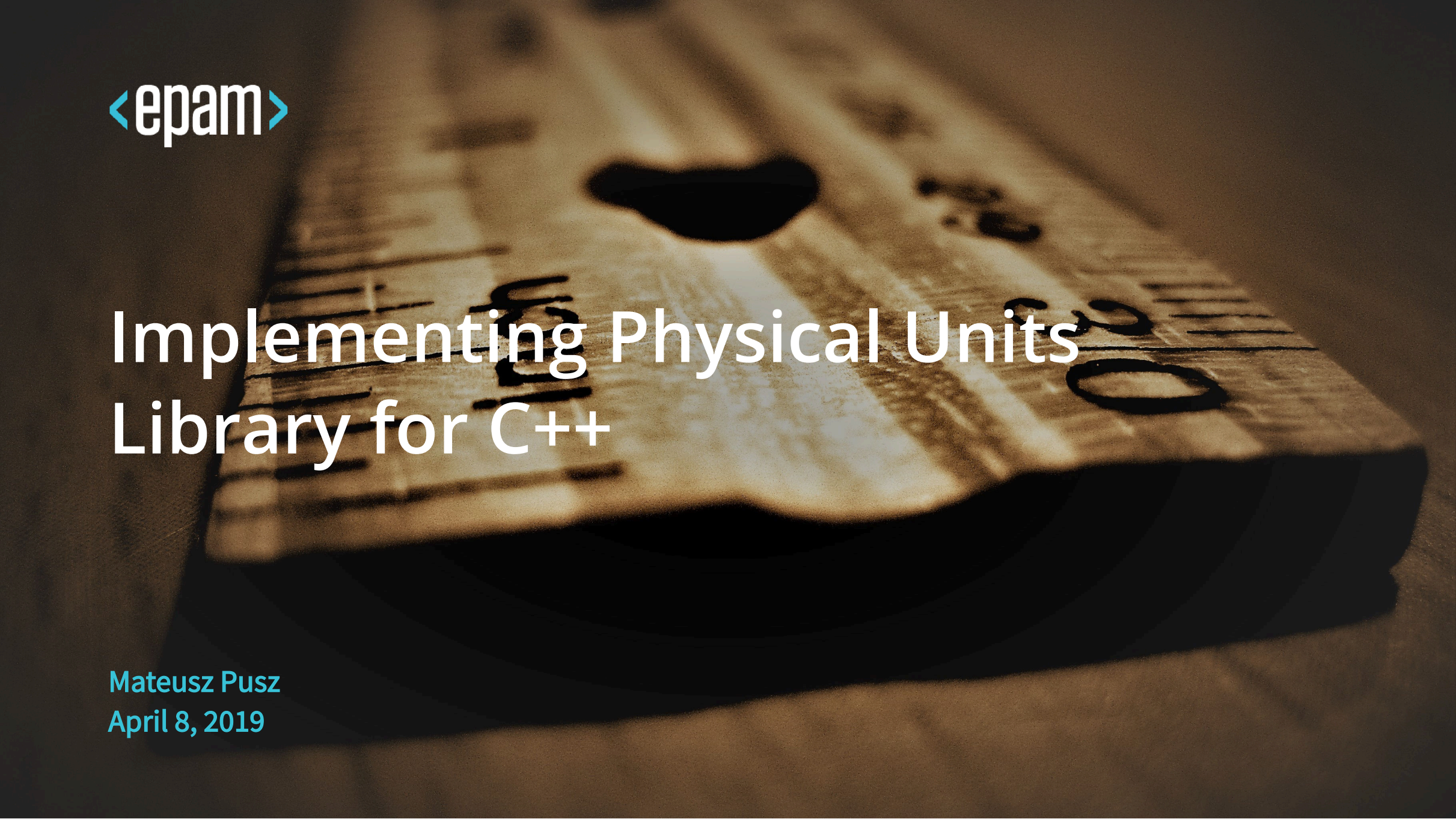


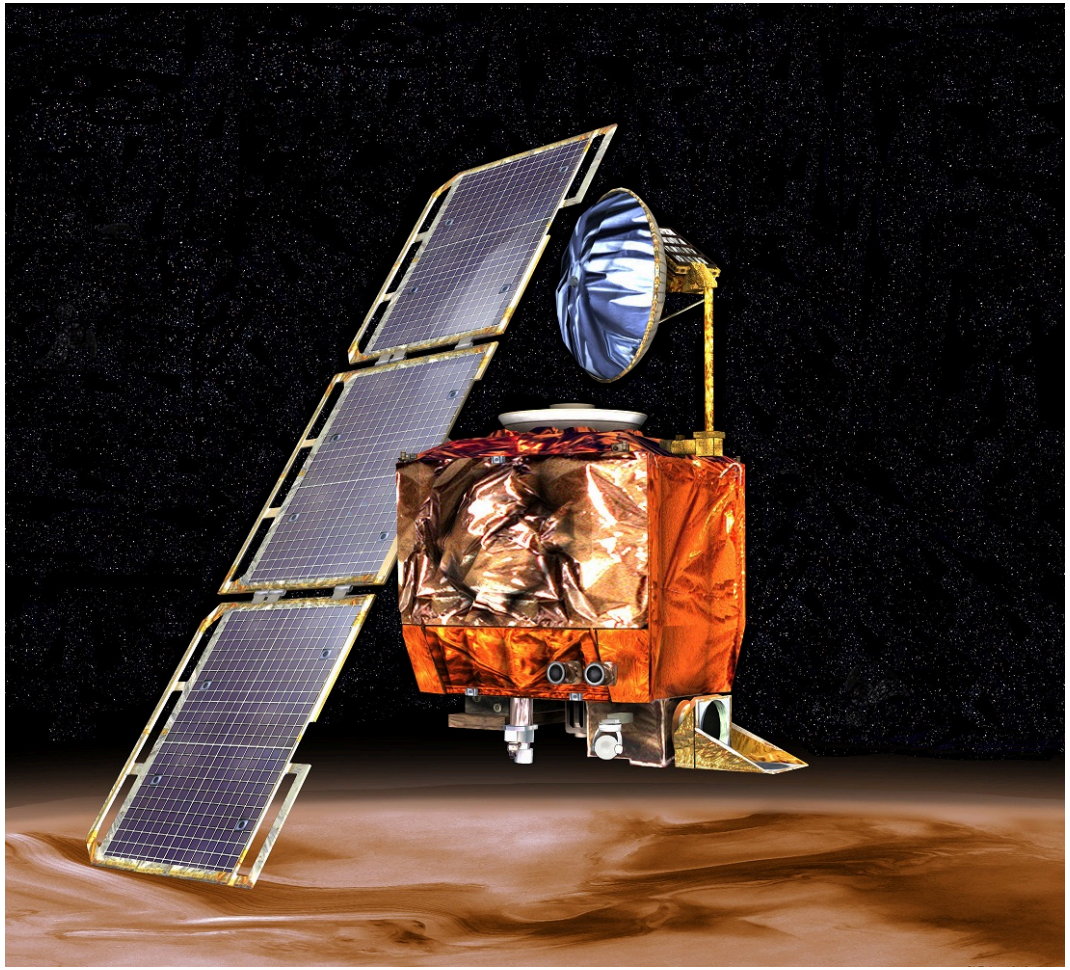
Implementing Physical Units Library for C++

Mateusz Pusz
April 8, 2019


?
!
Why?
Why Not?
Famous motivating example
?
!
Why?
Why Not?

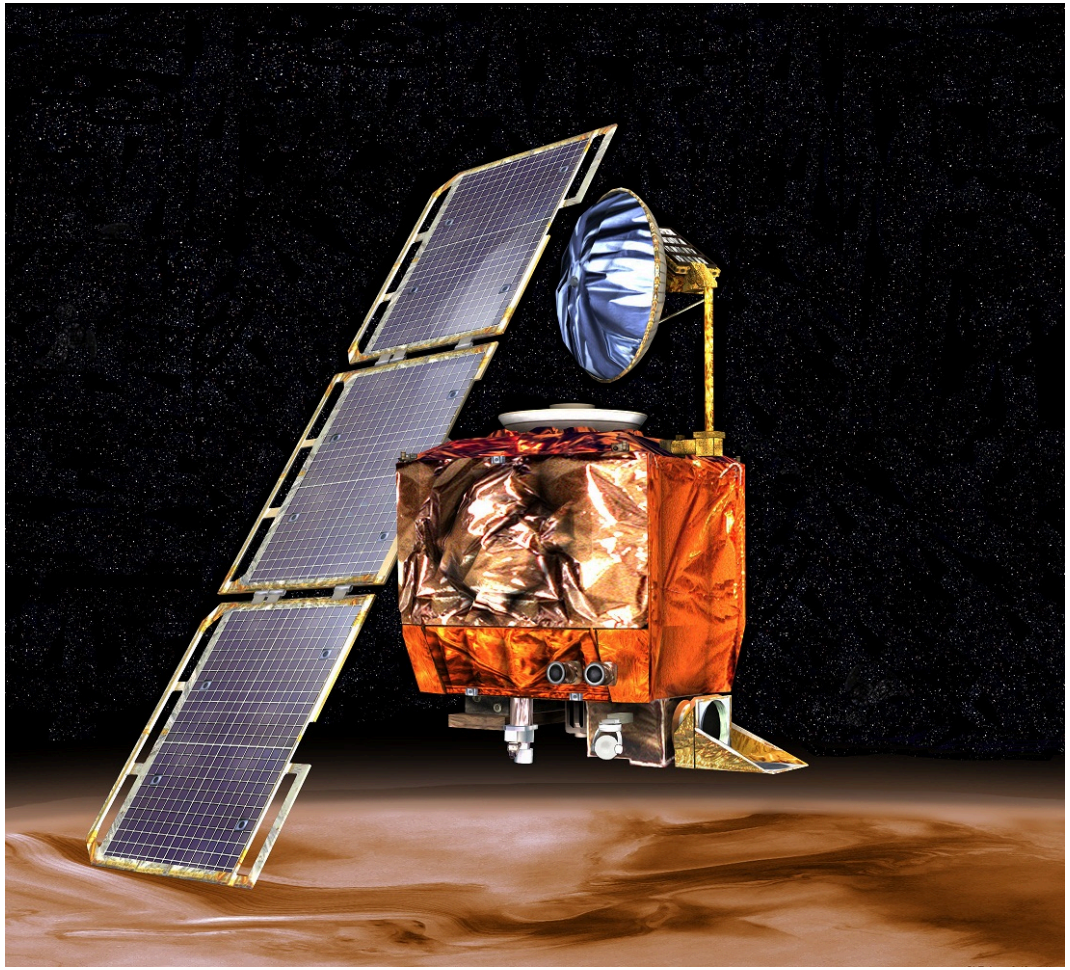
The Mars Climate Orbiter

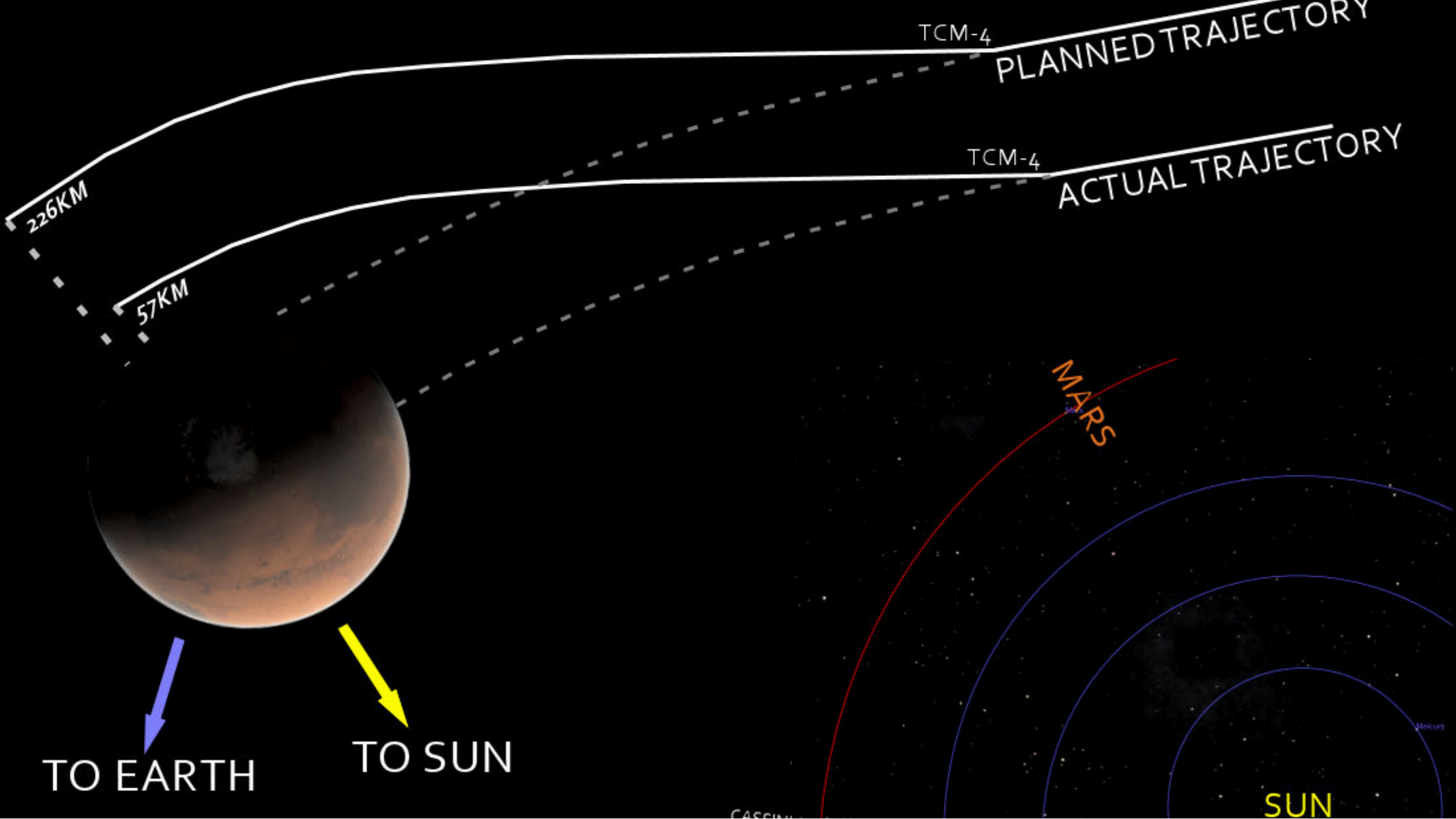
- **Robotic space probe** launched by NASA on December 11, 1998
- Project costs: **$327.6 million**



The Mars Climate Orbiter

- **Robotic space probe** launched by NASA on December 11, 1998
- Project costs: **$327.6 million**
- Mars Climate Orbiter began the planned *orbital insertion maneuver* on September 23, 1999 at 09:00:46 UTC




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

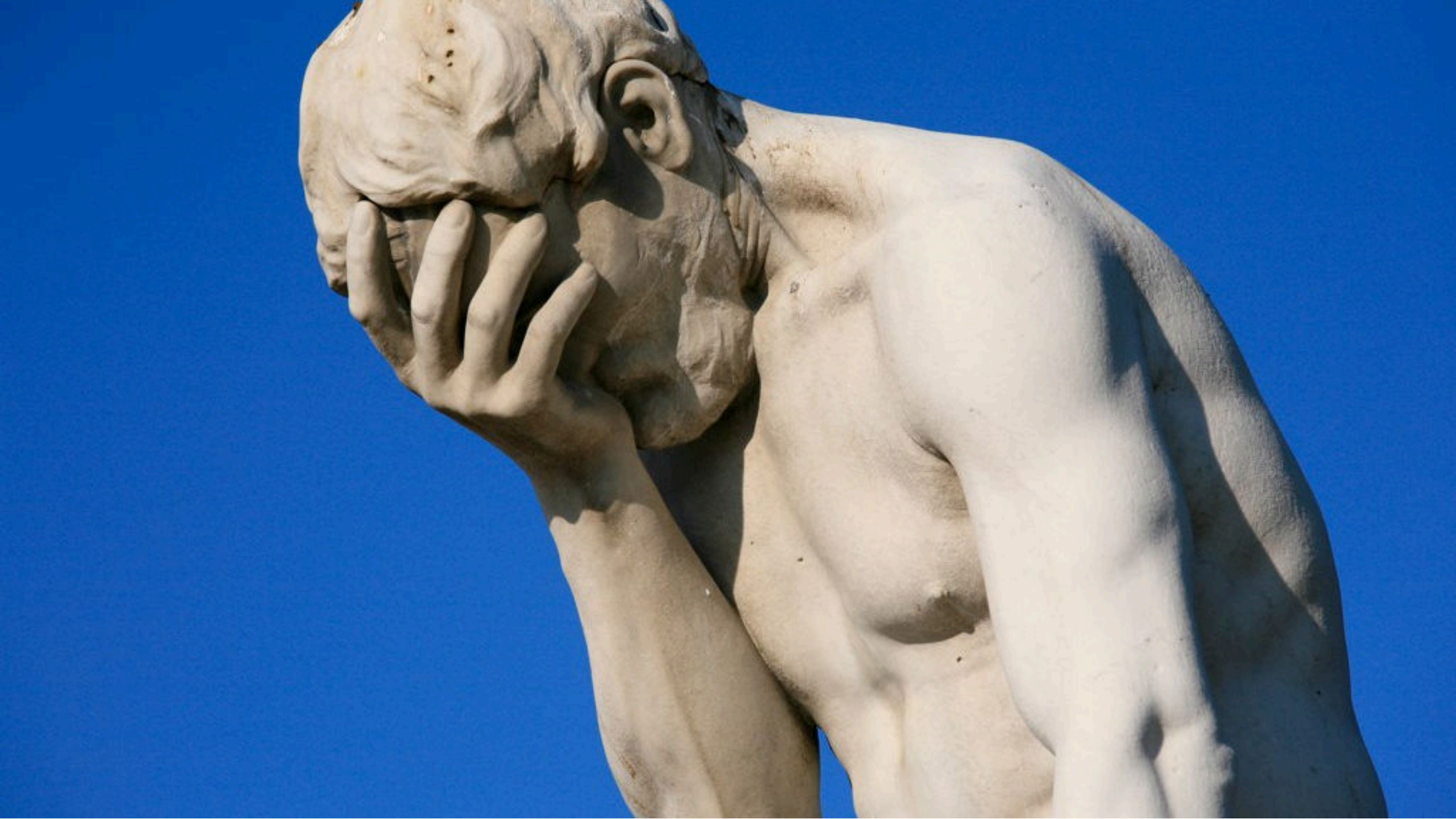
What went wrong?

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 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!

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

Existing solutions

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

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

- 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: 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);
```

```
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";
```

`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

```
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: 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

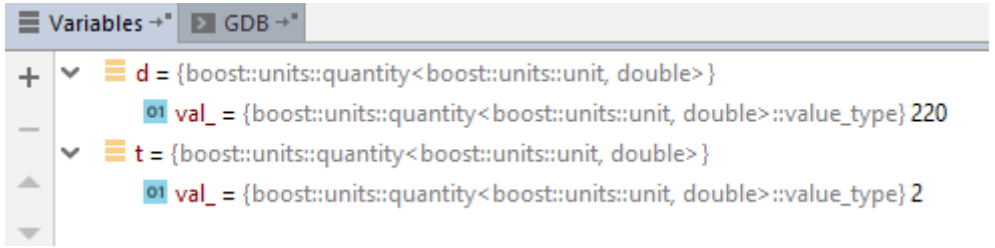
- do not influence the overload resolution process
- for some compilers do not provide enough context

User experience: 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
```

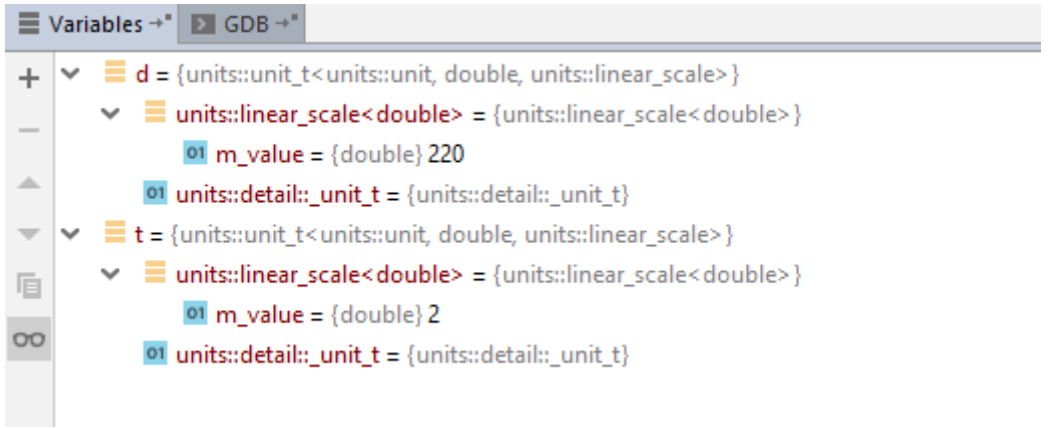
```
Breakpoint 1, (anonymous namespace)::avg_speed<boost::units::heterogeneous_system<boost::units::heterogeneous_system_impl
<boost::units::list<boost::units::heterogeneous_system_dim<boost::units::si::meter_base_unit, boost::units::static_rational<1> >,
boost::units::dimensionless_type>, boost::units::list<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: 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: 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: 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, (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;
```

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, 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

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

- 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 part of the C++ Standard Library

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 enity { /* ... */ };

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;
```

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;
```

EXAMPLE

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

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

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).

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

```
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** 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 = typename type_identity<T>::type;
```

Upcasting traits

```
template<typename T>
struct upcasting_traits : std::type_identity<T> {};

template<typename T>
using upcasting_traits_t = typename upcasting_traits<T>::type;
```

Upcasting traits

```
template<typename T>
struct upcasting_traits : std::type_identity<T> {};

template<typename T>
using upcasting_traits_t = typename upcasting_traits<T>::type;
```

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

Upcasting traits

```
template<typename T>
struct upcasting_traits : std::type_identity<T> {};

template<typename T>
using upcasting_traits_t = typename upcasting_traits<T>::type;
```

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

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

Upcasting traits

```
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 traits

```
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>> {};
```

```
using base = typename dimension_velocity::base_type;

template<>
struct upcasting_traits<base> : std::type_identity<dimension_velocity> {};
```

Upcasting traits

```
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>> {};
```

```
using base = typename dimension_velocity::base_type;

template<>
struct upcasting_traits<base> : std::type_identity<dimension_velocity> {};
```

TBD: Things to consider is to replace **`upcasting_traits`** with CRTP

Upcasting traits

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>
```

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

```
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
```

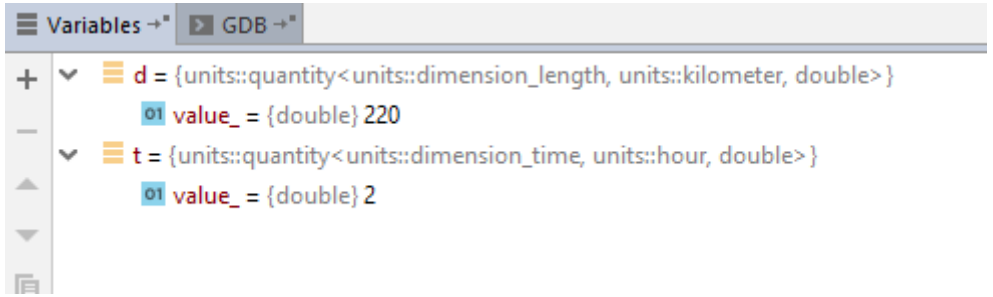
Probably it will be improved even more with time...

User experience: Debugging

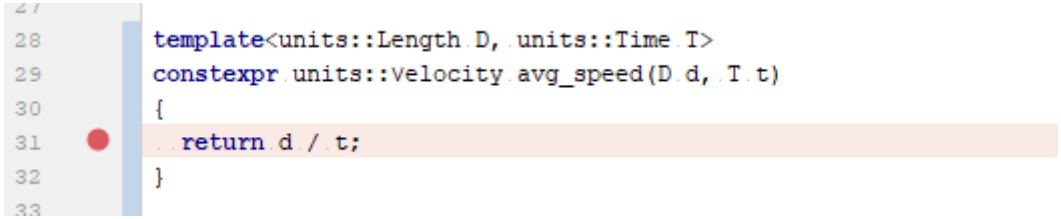
```
28  template<units::Length D, units::Time T>
29  constexpr units::Velocity avg_speed(D d, T t)
30  {
31      return d / t;
32  }
```

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, (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;
```

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";
```

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 length_miles = length<mile, Rep>;
```

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

Contracts

```
template<Dimension D, Unit U, Number Rep>
    requires std::experimental::ranges::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::experimental::ranges::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::experimental::ranges::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>;
    Expects(rhs != rhs_type(0));
    // ...
  }
};
```

Contracts

```
template<Dimension D, Unit U, Number Rep>
    requires std::experimental::ranges::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::experimental::ranges::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::experimental::ranges::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)]]
  {
    // ...
  }
};
```

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 ;-)

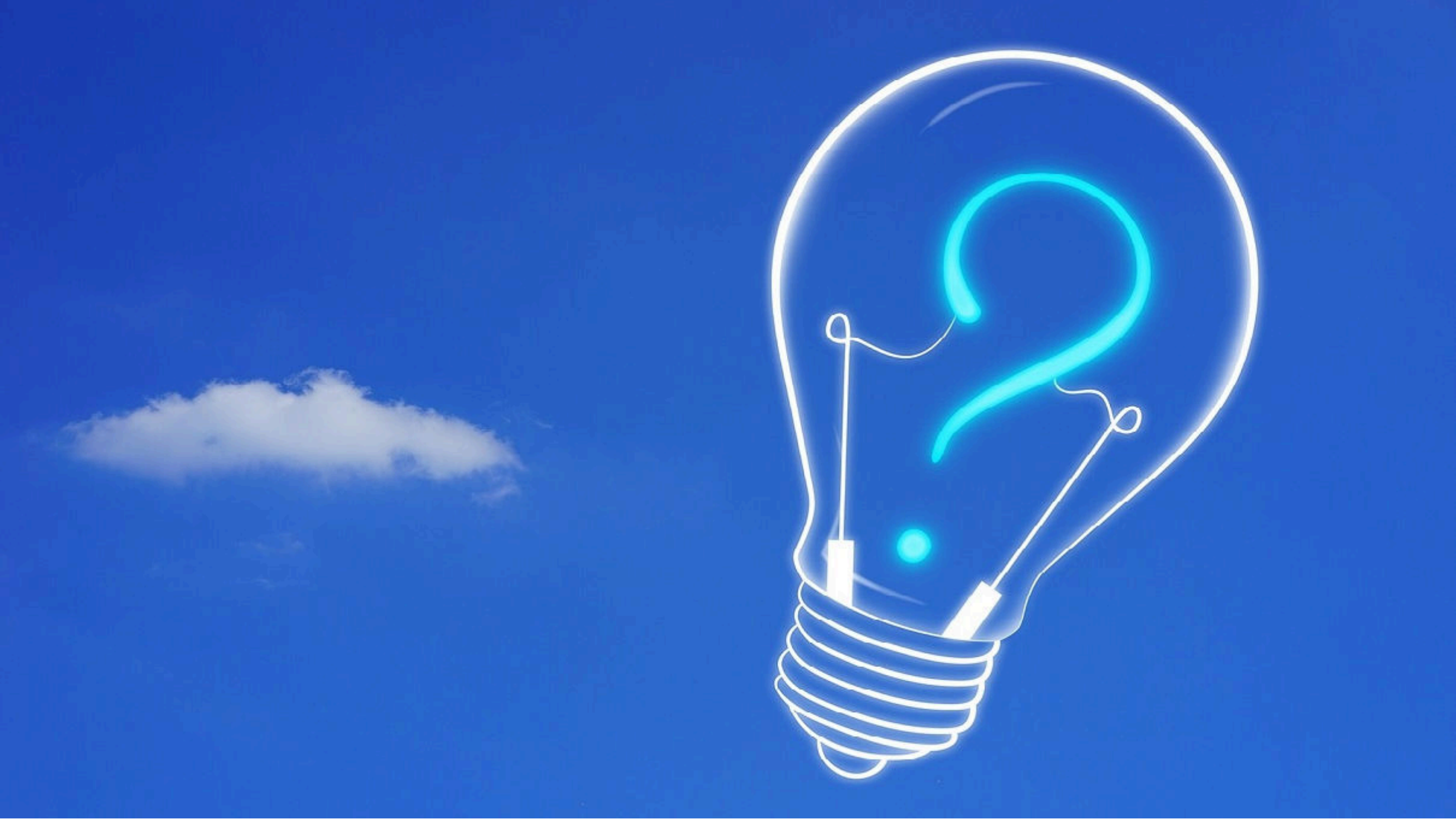
Let's join forces!

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)