ADAPT

BY ZED

VIRTUALCONFERENCE // NOVEMBER2020 // PRESENTED BY EPAM

ADAPT BY ZED

High-level Abstractions, Safety, or Performance

Mateusz Pusz

VIRTUALCONFERENCE // NOVEMBER2020 // PRESENTED BY EPAM

Who am I?

Mateusz Pusz

Associate Principal Software Engineer
Head of the C++ Competency Center
Head of EPAM Global C++ Community

Modern C++ Evangelist

- Hacking C++ for more than 15 years for fun and living
- Trainer, coach, mentor, consultant, and conference speaker
- Active voting member and contributor of the ISO C++ Committee (WG21)
- MISRA C++ member
- Mainly interested in code performance, low latency, safety, and maintainability

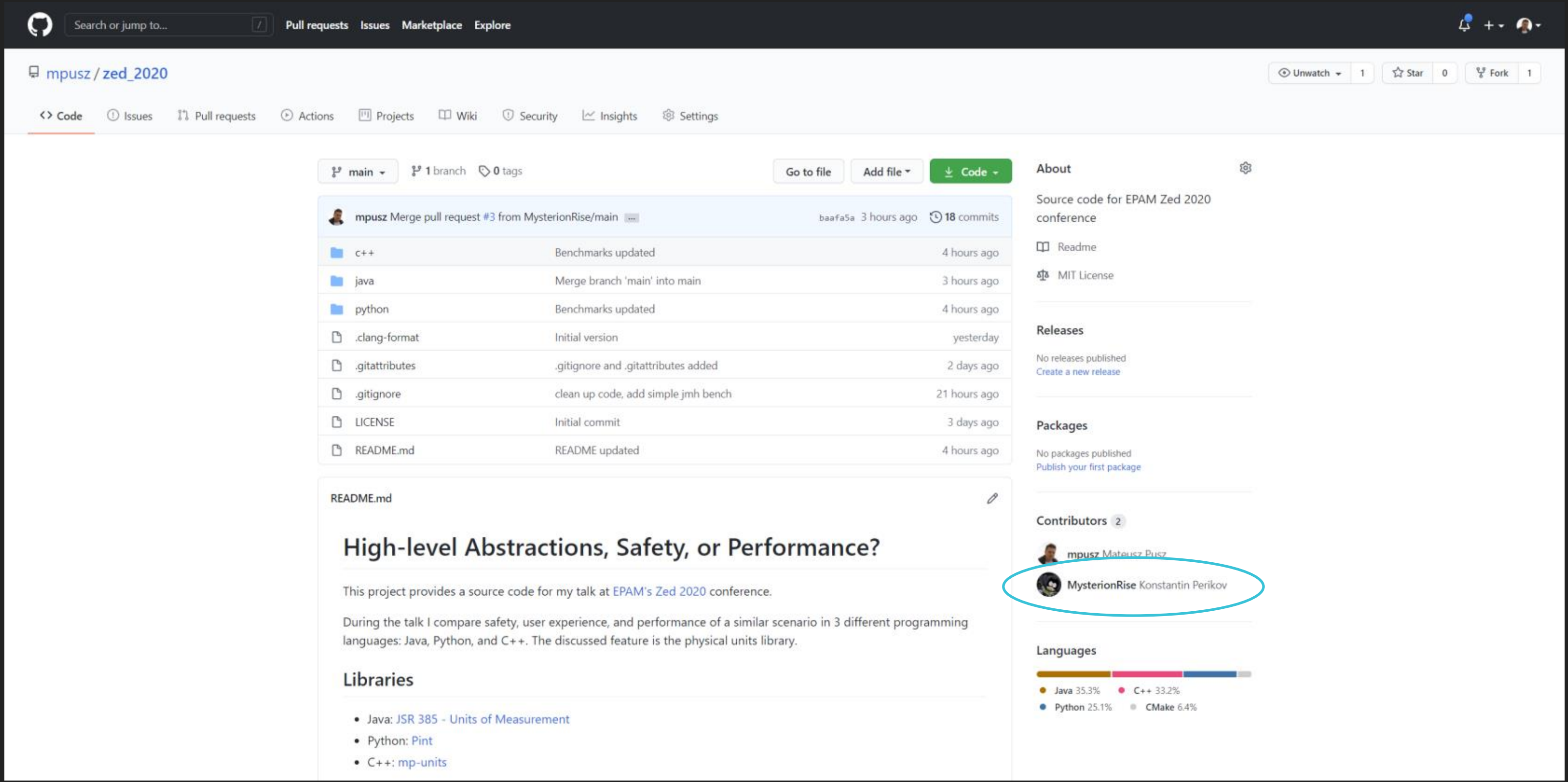
EPAM ENGINEERING DNA

I am not claiming that C++ is the best programming language out there. Each of the programming languages has its use and areas of applicability where it proves the best. Let us not start the holy wars here 😉

GitHub

https://github.com/mpusz/zed_2020

ADAPT BY ZED

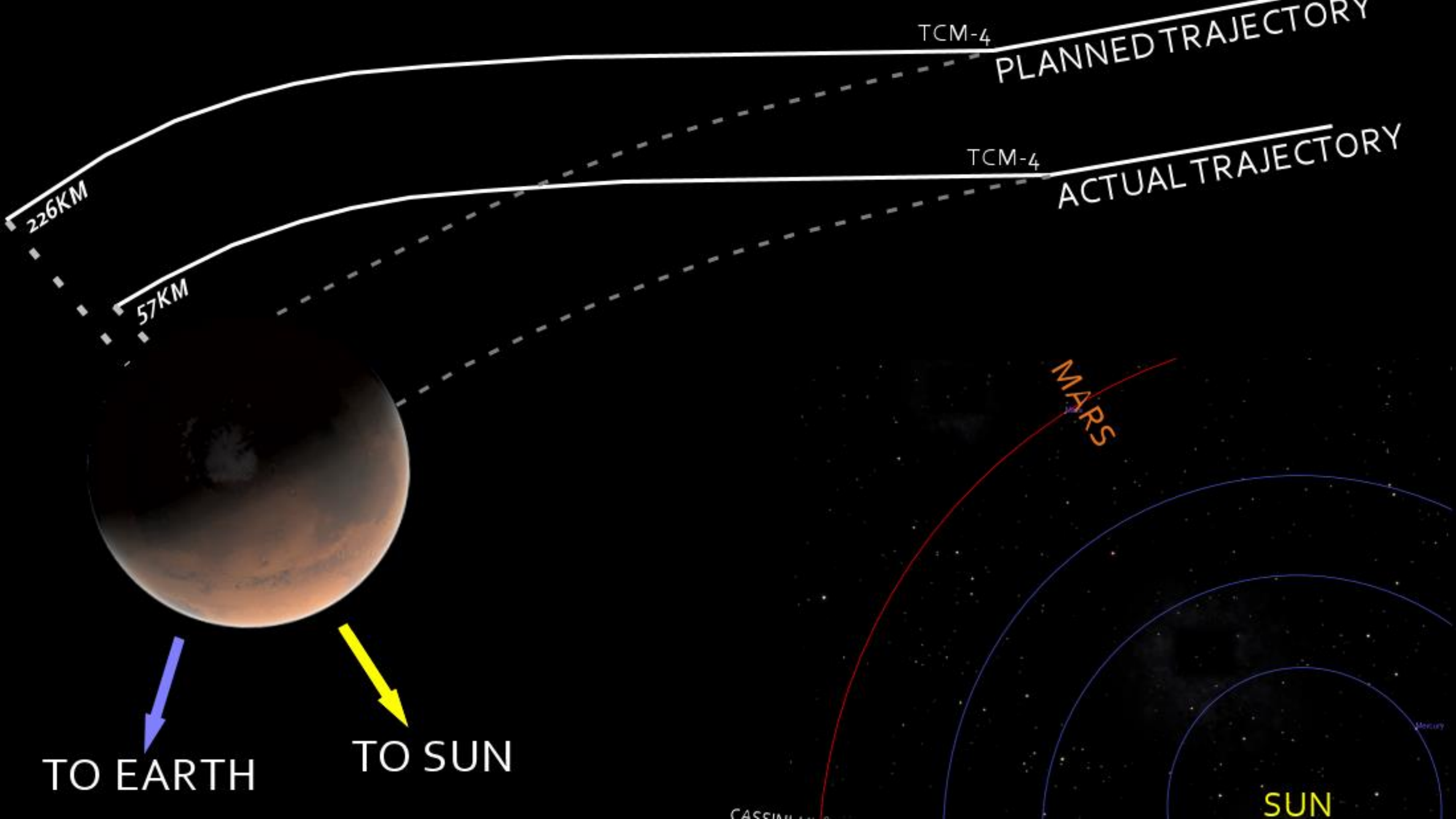


Agenda

1. **Motivation**
2. High-level abstractions
3. Safety
4. Efficiency

The Mars Climate Orbiter

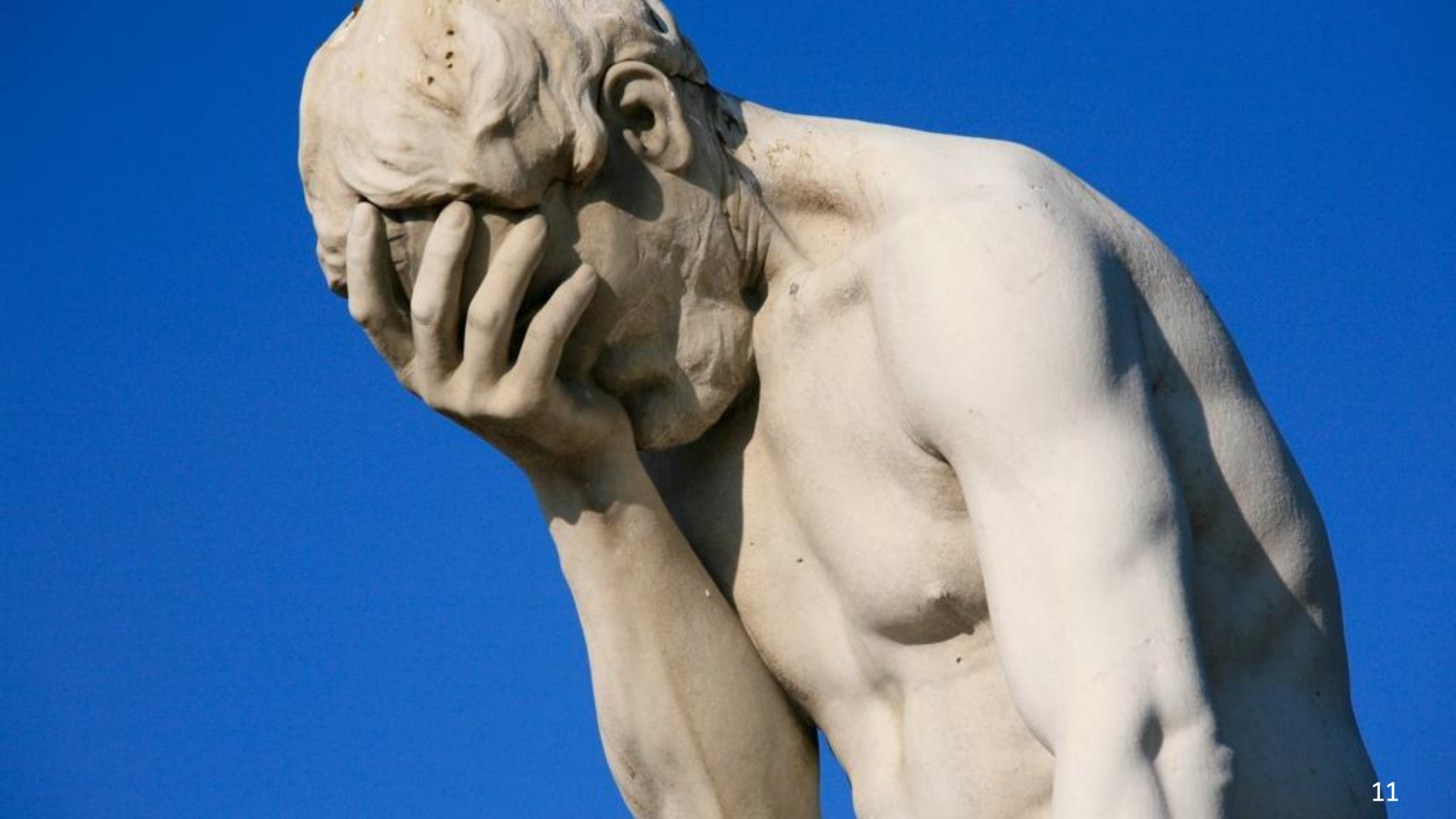
- Robotic space probe launched by NASA on December 11, 1998
- Project costs: $327.6 million
  - spacecraft development: $193.1 million
  - launching it: $91.7 million
  - mission operations: $42.8 million
- Mars Climate Orbiter began the planned orbital insertion maneuver on September 23, 1999 at 09:00:46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



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

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



A long time ago in a galaxy far far away... ADAPT BY ZED

Tactical Flight Computer

ADAPT BY ZED

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


```
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);
```

Code we shouldn't write


```
// 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)));
}
```

Now it is more important then ever

Agenda

1. Motivation
2. **High-level abstractions**
3. Safety
4. Efficiency

Comparison Scenario

- Implement avg_speed function that takes length and time arguments and returns speed in the unit derived from the units of function arguments
- Calculate avg_speed(220 km, 2 h) and print the result in km/h and m/s
- Calculate avg_speed(140 mi, 2 h) and print the result in mi/h and m/s

Pint

- Python package to define, operate and manipulate physical quantities
- It allows arithmetic operations between them and conversions from and to different units
- It is distributed with a comprehensive list of physical units, prefixes and constants.

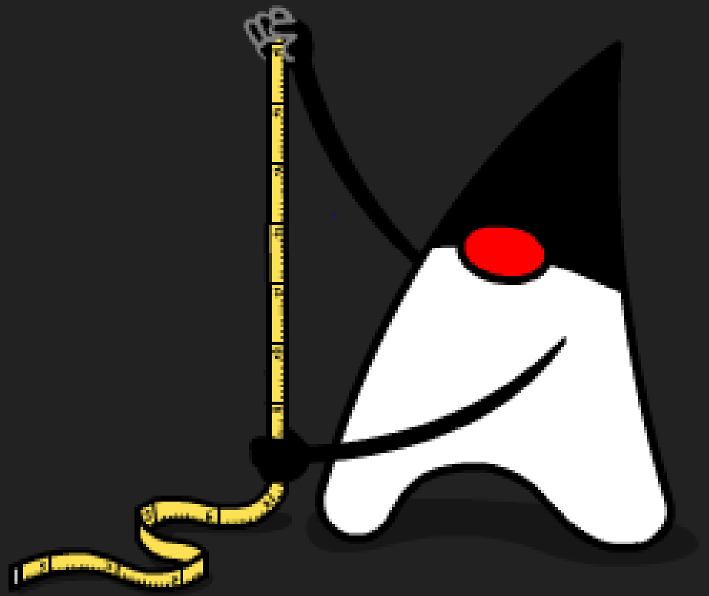
Pint


```
ureg = UnitRegistry()

@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  speed = d / t
  if not speed.check('[speed]'):
    raise RuntimeError("Not a [speed] dimension")
  return speed

s1 = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
s2 = avg_speed(140 * ureg.mile, 2 * ureg.hour)

def print_si(q):
  print('{:~P}'.format(q))

print_si(s1)
print_si(s2)
print_si(s1.to('metre/second’))
print_si(s2.to_base_units())
```

```
110.0 km/hr
70.0 mi/hr
30.555555555555557 m/s
31.292800000000003 m/s
```

JSR 385

- The Unit of Measurement API
- Provides a set of Java language programming interfaces for handling units and quantities
  - checking of unit compatibility
  - expression of a quantity in various units
  - arithmetic operations on units



JSR 385


```
public static Quantity<Speed> avg_speed(Quantity<Length> length,
                                        Quantity<Time> time) throws ClassCastException {
  return length.divide(time).asType(Speed.class);
}


final Quantity<Speed> s1 = avg_speed(Quantities.getQuantity(220., KILO(Units.METRE)),
                                     Quantities.getQuantity(2., Units.HOUR));
final Quantity<Speed> s2 = avg_speed(Quantities.getQuantity(140., MILE),
                                     Quantities.getQuantity(2., Units.HOUR));
System.out.println(s1);
System.out.println(s2);
System.out.println(s1.to(Units.METRE_PER_SECOND));
System.out.println(s2.toSystemUnit());
-----------------------------------------------------------------------------------------

 110 km/h
 70 mi/h
 30.5555555555555555555555555555556 m/s
 31.2928 m/s
```

mp-units

- Modern C++ library
- Provides compile-time dimensional analysis and unit/quantity manipulation
  - no runtime execution or memory storage space cost is introduced
- Support for quantities and units for arbitrary unit system models and arbitrary value types
- Planned for the ISO standardization as a part of the C++23/26

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

Speed auto s1 = avg_speed(si::length<si::kilometre>(220), si::time<si::hour>(2));
Speed auto s2 = avg_speed(si::length<si::international::mile>(140), si::time<si::hour>(2));
```

mp-units


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

Speed auto s1 = avg_speed(220_q_km, 2_q_h);
Speed auto s2 = avg_speed(140_q_mi, 2_q_h);

std::cout << s1 << '\n';
std::cout << s2 << '\n';
std::cout << quantity_cast<si::metre_per_second>(s1) << '\n';
std::cout << quantity_cast<si::dim_speed::coherent_unit>(s2) << '\n';
```

```
110 km/h
70 mi/h
30.5556 m/s
31.2928 m/s
```

Agenda

1. Motivation
2. High-level abstractions
3. **Safety**
4. Efficiency

Comparison Scenario

- Ensure that for reordered arguments avg_speed(2 h, 220 km) returns an error
- Ensure that an error is reported when avg_speed returns the result of an invalid calculation
    - function multiplies the arguments instead of dividing them
    - the result is not a quantity of speed

Pint – reordered arguments


```python
@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  speed = d / t
  if not speed.check('[speed]'):
    raise RuntimeError("Not a [speed] dimension")
  return speed

s1 = avg_speed(2 * ureg.hour, 220 * ureg.kilometer)
```

```
Traceback (most recent call last):
  File "safety_1.py", line 13, in <module>
    s1 = avg_speed(2 * ureg.hour, 220 * ureg.kilometer)
  File "/home/mpusz/.local/lib/python3.8/site-packages/pint/registry_helpers.py", line 350, in wrapper
    raise DimensionalityError(value, "a quantity of", val_dim, dim)
pint.errors.DimensionalityError: Cannot convert from '2 hour' ([time]) to 'a quantity of' ([length])
```

Runtime Error

Pint – invalid computation


```python
@ureg.check('[length]', '[time]')
def avg_speed(d, t):
  speed = d * t
  if not speed.check('[speed]'):
    raise RuntimeError("Not a [speed] dimension")
  return speed

s1 = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
```

```
Traceback (most recent call last):
  File "safety_2.py", line 14, in <module>
    s1 = avg_speed(220 * ureg.kilometer, 2 * ureg.hour)
  File "/home/mpusz/.local/lib/python3.8/site-packages/pint/registry_helpers.py", line 351, in wrapper
    return func(*args, **kwargs)
  File "safety_2.py", line 10, in avg_speed
    raise RuntimeError("Not a [speed] dimension")
RuntimeError: Not a [speed] dimension
```

Runtime Error

JSR 385 – reordered arguments

ADAPT BY ZED

```
public static Quantity<Speed> avg_speed(Quantity<Length> length,
                                        Quantity<Time> time) throws ClassCastException {
  return length.divide(time).asType(Speed.class);
}

public static void main(String[] args) {
  final Quantity<Speed> s = avg_speed(Quantities.getQuantity(2., Units.HOUR),
                                      Quantities.getQuantity(220., KILO(Units.METRE)));
}
```

```
[ERROR] Safety_1.java:[41,31] incompatible types: inference variable Q has incompatible equality constraints
javax.measure.quantity.Length,javax.measure.quantity.Time
```

Compile-time Error

JSR 385 – invalid computation

ADAPT BY ZED

```java
public static Quantity<Speed> avg_speed(Quantity<Length> length,
                                        Quantity<Time> time) throws ClassCastException {
  return length.multiply(time).asType(Speed.class);
}

public static void main(String[] args) {
  final Quantity<Speed> s = avg_speed(Quantities.getQuantity(220., KILO(Units.METRE)),
                                      Quantities.getQuantity(2., Units.HOUR));
}
```

```
Exception in thread "main" java.lang.ClassCastException: The unit: km·h is not compatible with quantities of type
interface javax.measure.quantity.Speed
        at tech.units.indriya.AbstractUnit.asType(AbstractUnit.java:277)
        at tech.units.indriya.AbstractUnit.asType(AbstractUnit.java:89)
        at tech.units.indriya.AbstractQuantity.asType(AbstractQuantity.java:337)
        at tech.units.indriya.AbstractQuantity.asType(AbstractQuantity.java:114)
        at zed2020.Safety_2.avg_speed(Safety_2.java:37)
        at zed2020.Safety_2.main(Safety_2.java:41)
```

Runtime Error

mp-units – reordered arguments


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

Speed auto s1 = avg_speed(2_q_h, 220_q_km);
```

```
avg_speed.cpp: In function ‘int main()’:
avg_speed.cpp:45:44: error: use of function ‘constexpr Speed auto avg_speed(auto:16, auto:17) [with auto:16 =
units::quantity<units::physical::si::dim_time, units::physical::si::hour, long int>; auto:17 =
units::quantity<units::physical::si::dim_length, units::physical::si::kilometre, long int>]’ with unsatisfied
constraints
    45 |   Speed auto s1 = avg_speed(2_q_h, 220_q_km);
       |                                           ^
avg_speed.cpp:29:16: note: declared here
    29 | constexpr Speed auto avg_speed(Length auto d, Time auto t)
       |                      ^~~~~~~~~
```

Compile-time Error

mp-units – invalid computation


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

Speed auto s1 = avg_speed(220_q_km, 2_q_h);
```

```
In instantiation of ‘constexpr auto [requires units::physical::Speed<<placeholder>, >] avg_speed(auto:16, auto:17)
[with auto:16 = units::quantity<units::physical::si::dim_length, units::physical::si::kilometre, long int>; auto:17
= units::quantity<units::physical::si::dim_time, units::physical::si::hour, long int>]’:
avg_speed.cpp:42:44:   required from here
avg_speed.cpp:32:14: error: deduced return type does not satisfy placeholder constraints
    32 |   return d * t;
       |            ^
```

Compile-time Error

Agenda

1. Motivation
2. High-level abstractions
3. Safety
4. **Efficiency**

Comparison Scenario

- Benchmark the following scenarios both for operations on fundamental/primitive types and on high-level quantity abstractions
  - Arithmetic - create quantities of length and time and divide them to obtain speed
  - Scaling - create a quantity of speed and convert the unit from km/h to m/s

Pint

Pint can impose a significant performance overhead on computationally-intensive problems. The following are some suggestions for getting the best performance.

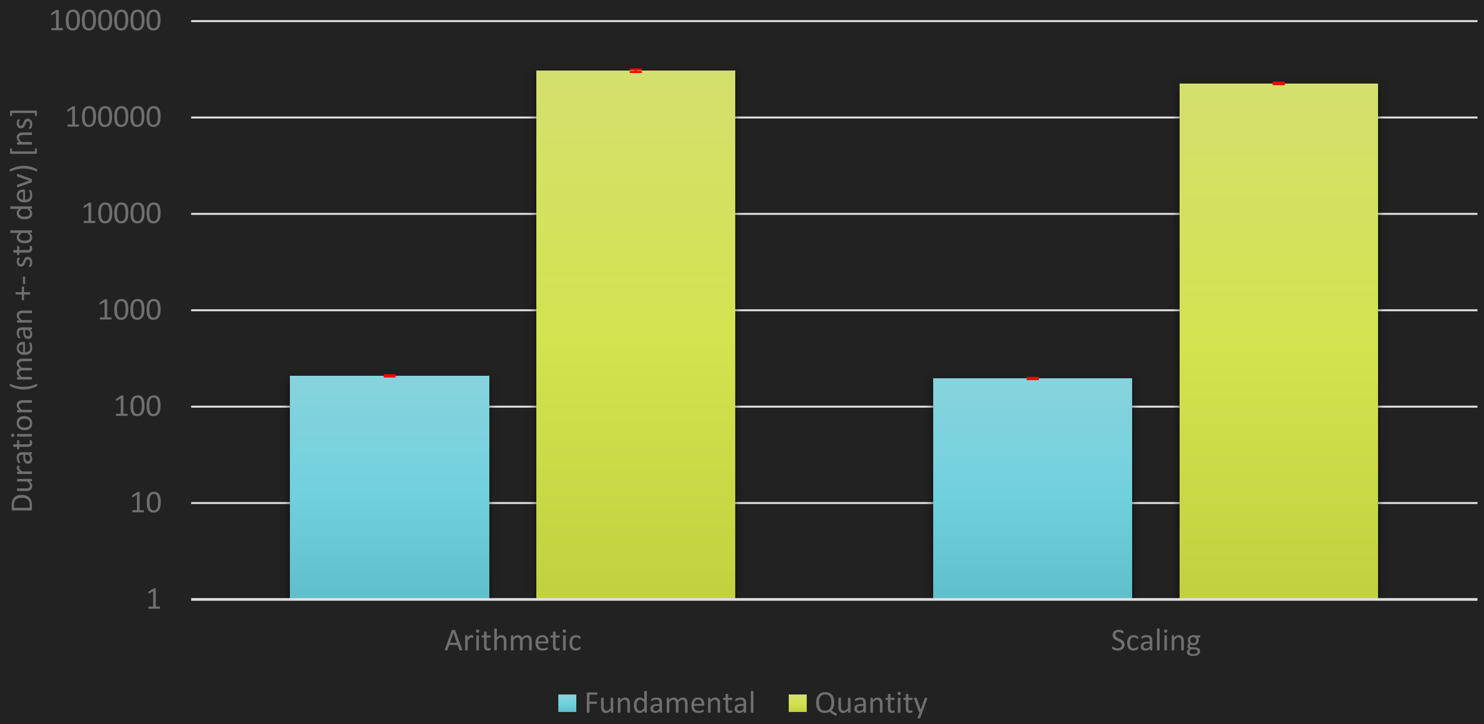
Use magnitudes when possible

It's significantly faster to perform mathematical operations on magnitudes (even though your'e still using pint to retrieve them from a quantity object).

```
In [1]: from pint import UnitRegistry

In [2]: ureg = UnitRegistry()

In [3]: q1 =ureg('1m')

In [5]: q2=ureg('2m')

In [6]: %timeit (q1-q2)
100000 loops, best of 3: 7.9 µs per loop

In [7]: %timeit (q1.magnitude-q2.magnitude)
1000000 loops, best of 3: 356 ns per loop
```

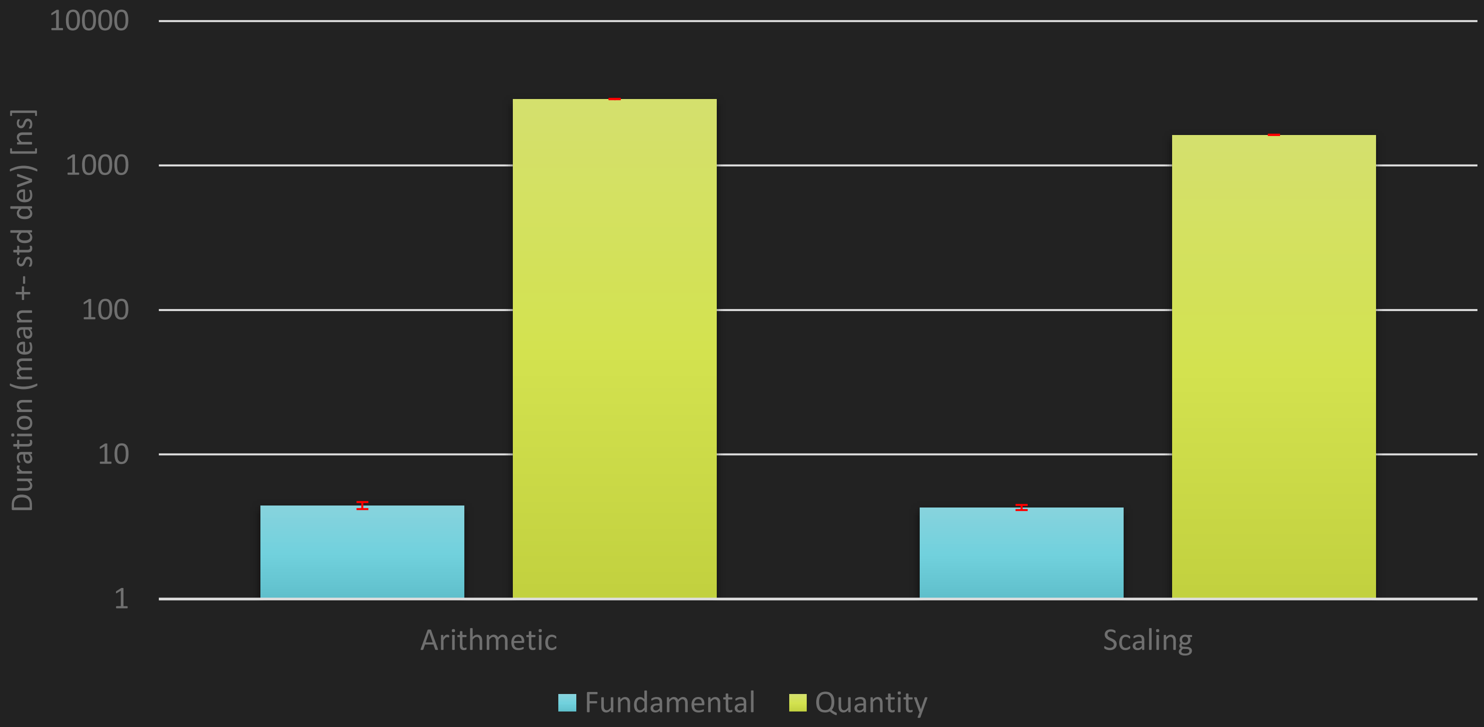
Bear in mind that altering computations like this **loses the benefits of automatic unit conversion**, so use with care.

Pint

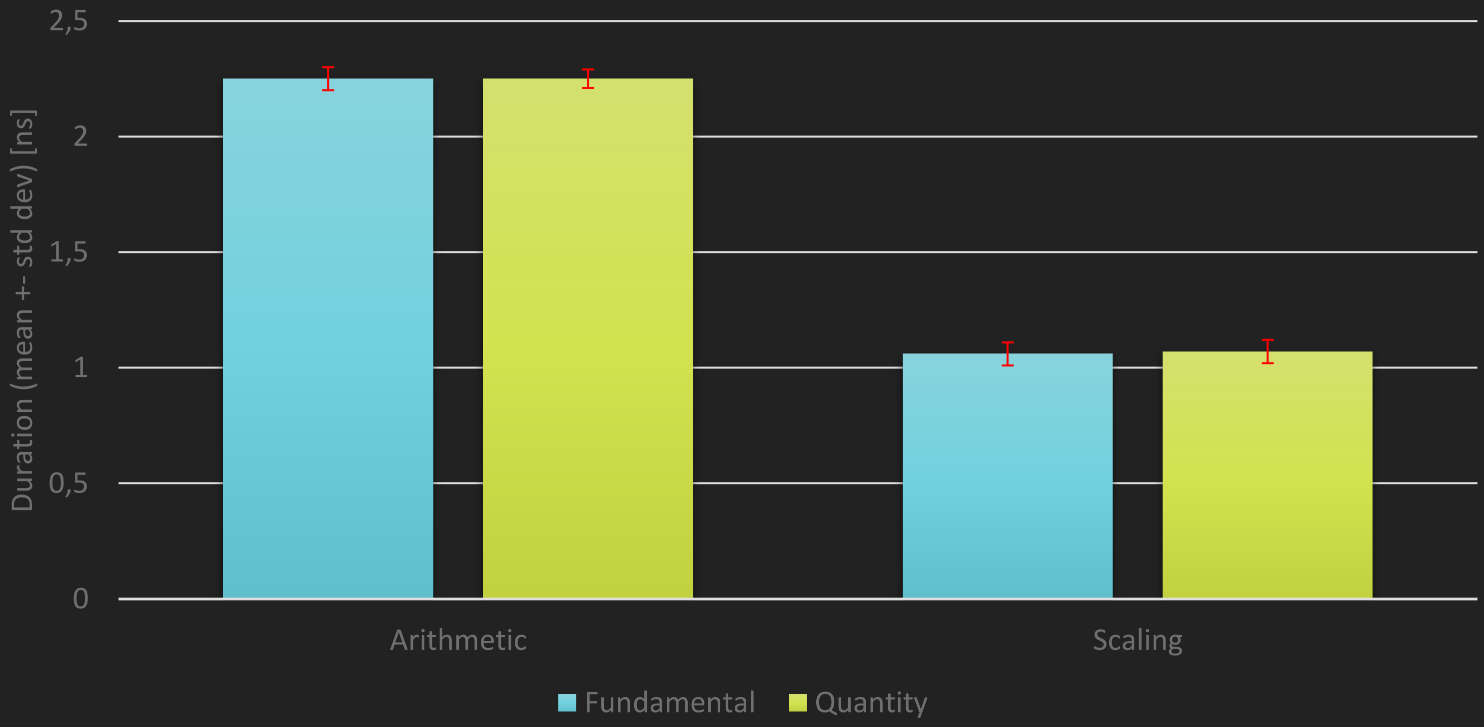


Duration (mean +- std dev) [ns]
1000000
100000
10000
1000
100
10
1
Arithmetic
Scaling
Fundamental
Quantity

JSR 385


10000
1000
100
10
1
Duration (mean +- std dev) [ns]
Arithmetic
Scaling
Fundamental
Quantity

mp-units



Duration (mean +- std dev) [ns]
2,5
2
1,5
1
0,5
0
Arithmetic
Scaling
Fundamental
Quantity

```cpp
constexpr double avg_speed(double d, double t)
{
  return d / t;
}
```

```asm
avg_speed(...):
        divsd   xmm0, xmm1
        ret
```

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

```asm
avg_speed(...):
        divsd   xmm0, xmm1
        ret
```

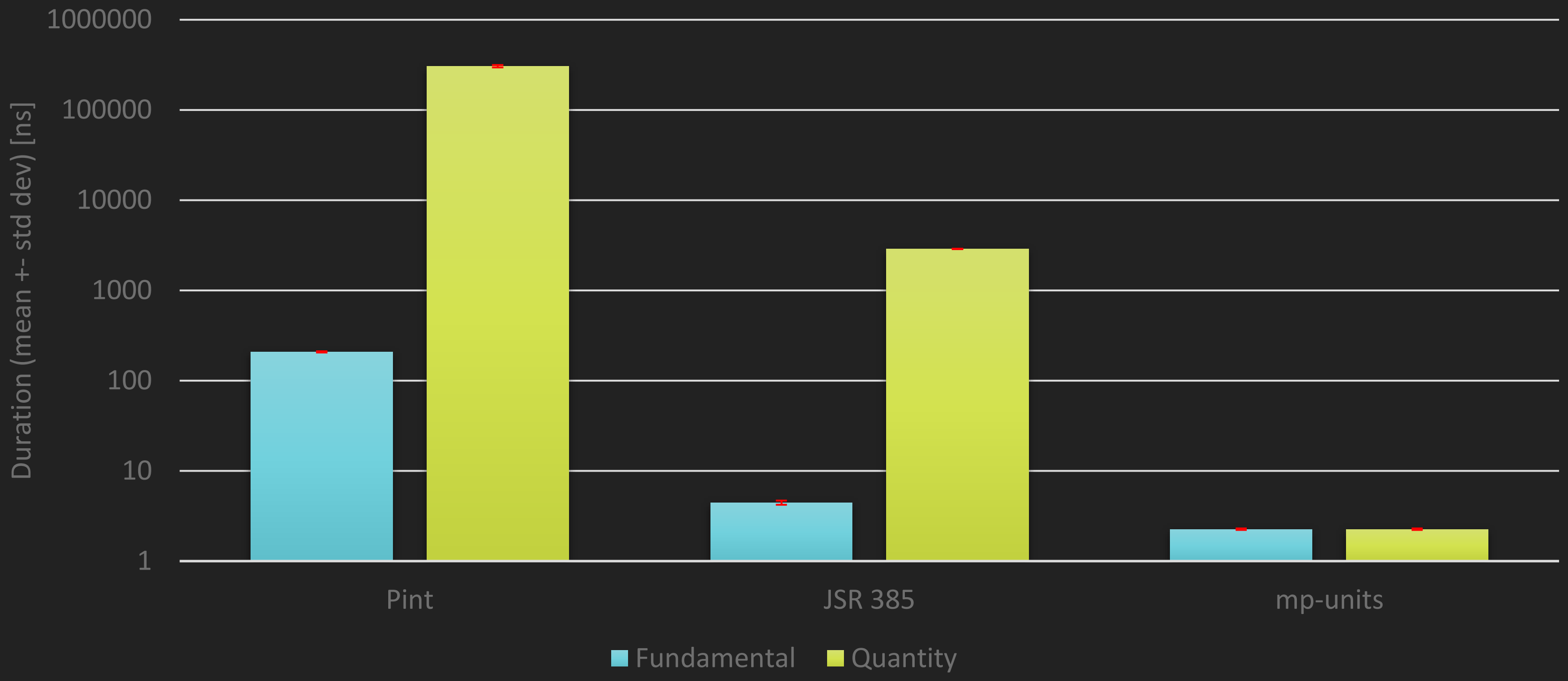
High-level abstractions without sacrificing runtime performance

Compile-time evaluation when possible

ADAPT BY ZED

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

// unit conversions
static_assert(1_q_h == 3600_q_s);
static_assert(1_q_km + 1_q_m == 1001_q_m);

// dimension conversions
static_assert(1_q_km / 1_q_s == 1000_q_m_per_s);
static_assert(2_q_km_per_h * 2_q_h == 4_q_km);
static_assert(2_q_km / 2_q_km_per_h == 1_q_h);
static_assert(2_q_m * 3_q_m == 6_q_m2);
static_assert(10_q_km / 5_q_km == 2);
static_assert(1000 / 1_q_s == 1_q_kHz);
```

Comparison (Arithmetic)



Efficiency is not just running fast or running bigger programs, it’s also running using less resources.

Bjarne Stroustrup, June 2011

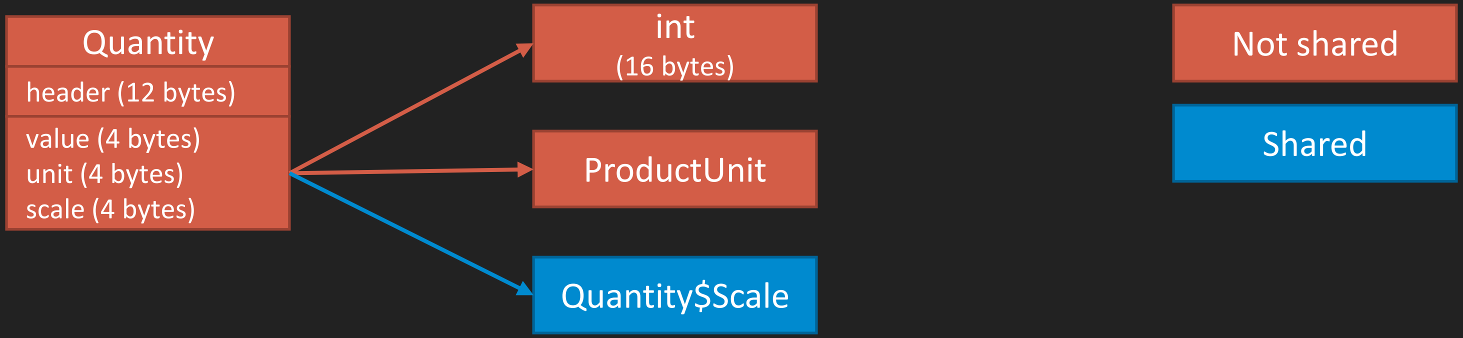
Pint

Quantity
_magnitude _units __used __handling

```
from pympler import asizeof

def test_data(count, min, max):
  values = default_rng().uniform(min, max, count)
  return list(map(lambda s: s * ureg.kilometre / ureg.hour, values))

print(asizeof.asizeof(test_data(10000, 40, 140)), "bytes")

6963520 bytes
```

1 Quantity ~ 700 bytes

JSR 385




```java
Quantity[] quantities = new Quantity[10000];
for (int i = 0; i < quantities.length; ++i) {
  quantities[i] = avg_speed(Quantities.getQuantity(Math.random() * 1000, KILO(Units.METRE)),
                            Quantities.getQuantity(Math.random() * 10, Units.HOUR));
}
System.out.println(RamUsageEstimator.sizeOf(quantities) + " bytes");
```

```
5001128 bytes
```

1 Quantity ~ 500 bytes

quantity<Dimension D, UnitOf<D> U, QuantityValue Rep>
Rep value

```
namespace si {

template<UnitOf<dim_length> U, QuantityValue Rep = double>
using length = quantity<dim_length, U, Rep>;

}

static_assert(sizeof(si::length<si::metre>) == sizeof(double));
static_assert(sizeof(si::length<si::metre, int>) == sizeof(int));
static_assert(sizeof(si::length<si::metre, std::int8_t>) == 1);
```

Modern C++ provides safety and high-level abstractions without sacrificing on efficiency.

ADAPT BY ZED

Thank you!