3-dimensions

June 3, 2024

1 Dimensional Analysis

Strong typedef and **customized suffix** enables to define specific types for different basic physics quantities, and for their multiples, but we quickly noticed derived units are missing, formed by powers, products or fractions of basic units. There is an illimited number of possible combinations.

1.1 A single class for all units

Rather than writing an infinite number of classes, we will use a single template class, which will include integer parameters to describe the dimension of each base units : 1. Mass, in kilograms, 2. Time, in seconds, 3. Length, in meters, 4. Thermodynamic temperature, in kelvins, 5. Electric current, in amperes, 6. Amount of substance, in moles, 7. Luminous Intensity, in candelas.

1.2 A simplified example

To introduce the idea with a simplified case, let's create a class template with 2 integers parameters, which correspond to the powers of both the `time` and `lenght` dimensions, and a generic operator `/` allowing us to compute a velocity or an acceleration :

[1]:
```
%%file tmp.dimensions.siunit-1.h

template< typename UnderlyingType, int s, int m >
class SiUnit
 {
  public :
    explicit constexpr SiUnit( UnderlyingType value ) : my_value{value} {}
    explicit constexpr operator UnderlyingType() const { return my_value ; }
    friend bool operator<=( SiUnit lhs, SiUnit rhs ) { return (lhs.
 ↪my_value<=rhs.my_value) ; }
  private :
    UnderlyingType my_value ;
 } ;
```

```
Writing tmp.dimensions.siunit-1.h
```

[2]:
```
%%file tmp.dimensions.siunit-2.h

template< typename UT, int s, int m >
std::ostream & operator<<( std::ostream & os, const SiUnit<UT,s,m> & obj )
```

```
 { return (os<<static_cast<UT>(obj)) ; }

template< typename UT, int s1, int m1, int s2, int m2 >
constexpr auto operator*( SiUnit<UT,s1,m1> lhs, SiUnit<UT,s2,m2> rhs )
 { return SiUnit<UT,s1+s2,m1+m2>(static_cast<UT>(lhs)*static_cast<UT>(rhs)) ; }

template< typename UT, int s1, int m1, int s2, int m2 >
constexpr auto operator/( SiUnit<UT,s1,m1> lhs, SiUnit<UT,s2,m2> rhs )
 { return SiUnit<UT,s1-s2,m1-m2>(static_cast<UT>(lhs)/static_cast<UT>(rhs)) ; }
```

```
Writing tmp.dimensions.siunit-2.h
```

[3]:
```
%%file tmp.dimensions.siunit-3.h

using Time = SiUnit<double,1,0> ;
constexpr Time operator "" _Se ( long double value )
 { return Time{static_cast<double>(value)} ; }
constexpr Time operator "" _Mi ( long double value )
 { return Time{static_cast<double>(value)*60} ; }
constexpr Time operator "" _H ( long double value )
 { return Time{static_cast<double>(value)*60*60} ; }

using Length = SiUnit<double,0,1> ;
constexpr Length operator "" _M ( long double value )
 { return Length{static_cast<double>(value)} ; }
constexpr Length operator "" _KM ( long double value )
 { return Length{static_cast<double>(value)*1000} ; }

using Speed = SiUnit<double,-1,1> ;
```

```
Writing tmp.dimensions.siunit-3.h
```

[4]:
```
%%file tmp.dimensions.cpp

#include <iostream>
#include "tmp.dimensions.siunit-1.h"
#include "tmp.dimensions.siunit-2.h"
#include "tmp.dimensions.siunit-3.h"

int main()
 {
  constexpr auto l { 28._KM } ;
  constexpr auto d { 39._Mi } ;
  constexpr Speed vmax = 50._KM/1._H ;
  constexpr Speed v = l/d ;
  std::cout<<"Max  speed : "<<vmax<<" m/s"<<std::endl ;
  std::cout<<"Mean speed : "<<v<<" m/s"<<std::endl ;
  if (v<=vmax) { std::cout<<"You drive safely :)"<<std::endl ; }
```

```
 else { std::cout<<"You drive too fast :("<<std::endl ; }
}
```

```
Writing tmp.dimensions.cpp
```

[5]:
```
!rm -f tmp.dimensions.exe && g++ -std=c++17 tmp.dimensions.cpp -o tmp.
↪dimensions.exe
```

[6]:
```
!./tmp.dimensions.exe
```

```
Max  speed : 13.8889 m/s
Mean speed : 11.9658 m/s
You drive safely :)
```

Note with our new "dimensionned" units, if we write `l*d` instead of `l/d` by mistake, we will produce a quantity with different type as velocity, and the compiler will detect the issue !

1.3 Last refinements

In our previous example, we can regret the heaviness of the expression `50._KM/1._H`, where we would simply like to say `50_KM/H`. As the use of `/` is not allowed in the suffix, we cannot create suffix `_KM/H`, but we still can go back to the creation of constants to make this writing valid. Furthermore, we can get rid of the `.` by overloading also the variant `unsigned long long` of suffix.

[7]:
```
%%file tmp.dimensions.siunit-3.h

using Time = SiUnit<double,1,0> ;

constexpr Time operator ""_Se ( long double value )
 { return Time{static_cast<double>(value)} ; }
constexpr Time operator ""_Mi ( long double value )
 { return Time{static_cast<double>(value)*60} ; }
constexpr Time operator ""_H ( long double value )
 { return Time{static_cast<double>(value)*60*60} ; }

constexpr Time operator ""_Se ( unsigned long long value )
 { return Time{static_cast<double>(value)} ; }
constexpr Time operator ""_Mi ( unsigned long long value )
 { return Time{static_cast<double>(value)*60} ; }
constexpr Time operator ""_H ( unsigned long long value )
 { return Time{static_cast<double>(value)*60*60} ; }

constexpr Time Se { 1._Se } ;
constexpr Time Mi { 1._Mi } ;
constexpr Time H { 1._H } ;
```

```
Overwriting tmp.dimensions.siunit-3.h
```

[8]:
```
%%file tmp.dimensions.siunit-4.h

using Length = SiUnit<double,0,1> ;

constexpr Length operator ""_M ( long double value )
 { return Length{static_cast<double>(value)} ; }
constexpr Length operator ""_KM ( long double value )
 { return Length{static_cast<double>(value)*1000} ; }

constexpr Length operator ""_M ( unsigned long long value )
 { return Length{static_cast<double>(value)} ; }
constexpr Length operator ""_KM ( unsigned long long value )
 { return Length{static_cast<double>(value)*1000} ; }

constexpr Length M { 1._M } ;
constexpr Length KM { 1._KM } ;
```

```
Writing tmp.dimensions.siunit-4.h
```

[9]:
```
%%file tmp.dimensions.siunit-5.h

using Speed = SiUnit<double,-1,1> ;
```

```
Writing tmp.dimensions.siunit-5.h
```

[10]:
```
%%file tmp.dimensions.cpp

#include <iostream>
#include "tmp.dimensions.siunit-1.h"
#include "tmp.dimensions.siunit-2.h"
#include "tmp.dimensions.siunit-3.h"
#include "tmp.dimensions.siunit-4.h"
#include "tmp.dimensions.siunit-5.h"

int main()
 {
  constexpr auto l { 28_KM } ;
  constexpr auto d { 39_Mi } ;
  constexpr Speed vmax = 50_KM/H ;
  constexpr Speed v = l/d ;
  std::cout<<"Max  speed : "<<vmax<<" m/s"<<std::endl ;
  std::cout<<"Mean speed : "<<v<<" m/s"<<std::endl ;
  if (v<=vmax) { std::cout<<"You drive safely :)"<<std::endl ; }
  else { std::cout<<"You drive too fast :("<<std::endl ; }
 }
```

```
Overwriting tmp.dimensions.cpp
```

[11]:
```
!rm -f tmp.dimensions.exe && g++ -std=c++17 tmp.dimensions.cpp -o tmp.
 ↪dimensions.exe
```

[12]:
```
!./tmp.dimensions.exe
```

```
Max  speed : 13.8889 m/s
Mean speed : 11.9658 m/s
You drive safely :)
```

1.4 Libraries to the rescue

It is quite complex to write classes which describe the complete international system of units. Rather try one of the available libraries : * Boost Units : the ancient one, robust, documented, but depend of other Boost libraries… * PhysUnits from Martin Moene : upgraded for C++11, popular, yet poorly documented. * units from Tony Pilz : newer, ligther. * units from Nick Holthaus: based on C++14, documented, active. * mp-units from Mateusz Pusz : leading edge, requires C++20, implements a proposal to the standardization committee, with a very detailed analysis of what exists.

The code below demonstrates the use of PhysUnits from Martin Moene.

[13]:
```
%%file tmp.dimensions.cpp

#include <iostream>

#include "phys/units/io.hpp"
#include "phys/units/quantity.hpp"

using namespace phys::units ;
using namespace phys::units::io;
using namespace phys::units::literals ;

int main()
 {
  constexpr auto l { 28_km } ;
  constexpr auto d { 39_min } ;
  constexpr quantity<speed_d> vmax = 50_km/hour ;
  constexpr quantity<speed_d> v = l/d ;
  std::cout<<"Max  speed : "<<vmax<<std::endl ;
  std::cout<<"Mean speed : "<<v<<std::endl ;
  if (v<=vmax) { std::cout<<"You drive safely :)"<<std::endl ; }
  else { std::cout<<"You drive too fast :("<<std::endl ; }
 }
```

```
Overwriting tmp.dimensions.cpp
```

[14]:
```
!rm -f tmp.dimensions.exe && g++ -I. -std=c++17 tmp.dimensions.cpp -o tmp.
 ↪dimensions.exe
```

[15]:
```
!./tmp.dimensions.exe
```

```
Max  speed : 13.8889 m/s
Mean speed : 11.9658 m/s
You drive safely :)
```

2 Questions ?

3 Exercise

Below, try to write one overload of the << operateur for `Velocity`, so to print the value in km/h.

[16]:
```
%%file tmp.dimensions.siunit-5.h

using Speed = SiUnit<double,-1,1> ;

// TO BE MODIFIED
std::ostream & operator<<( std::ostream & os, Speed v )
 {
  return (os<<static_cast<double>(v)<<" m/s") ;
 }
```

```
Overwriting tmp.dimensions.siunit-5.h
```

[17]:
```
%%file tmp.dimensions.cpp

#include <iostream>
#include "tmp.dimensions.siunit-1.h"
#include "tmp.dimensions.siunit-2.h"
#include "tmp.dimensions.siunit-3.h"
#include "tmp.dimensions.siunit-4.h"
#include "tmp.dimensions.siunit-5.h"

int main()
 {
  constexpr auto l { 28_KM } ;
  constexpr auto d { 39_Mi } ;
  constexpr Speed vmax = 50_KM/H ;
  constexpr Speed v = l/d ;
  std::cout<<"Max  speed : "<<vmax<<std::endl ;
  std::cout<<"Mean speed : "<<v<<std::endl ;
  if (v<=vmax) { std::cout<<"You drive safely :)"<<std::endl ; }
  else { std::cout<<"You drive too fast :("<<std::endl ; }
 }
```

```
Overwriting tmp.dimensions.cpp
```

[18]:
```
!rm -f tmp.dimensions.exe && g++ -std=c++17 tmp.dimensions.cpp -o tmp.
↪dimensions.exe
```

[19]:
```
!./tmp.dimensions.exe
```

```
Max  speed : 13.8889 m/s
Mean speed : 11.9658 m/s
You drive safely :)
```

4 Ressources & inspirations

SI * Wikipedia

Blogs * https://kaushikghose.wordpress.com/2018/05/06/c-dimensional-analysis/ * https://gmpreussner.com/research/dimensional-analysis-in-programming-languages