2-quantities

June 3, 2024

1 Multiple of units and user-defined litterals

1.1 Motivation

Strong typing enables to differantiate quantities, and give units to our variables, rather than using simples builtin types such as `int` or `double`.

However, when we consider the many units which are multiples one of another, such as meters and kilometers, we feel they are mostly **interoperable**. Rather than coding a different type for each multiple, let's introduce some kind of multipliers.

1.2 Old fashion constants

The simple legacy solution consist to define constants which are multiplied by the original literal values.

[1]:
```
template< typename UnderlyingType, typename TagType >
class StrongTypedef
 {
  public :
    explicit StrongTypedef( UnderlyingType value ) : my_value{value} {}
    explicit operator UnderlyingType() const { return my_value ; }
    UnderlyingType operator*( UnderlyingType value ) const { return␣
↪my_value*value ; }
    UnderlyingType operator/( UnderlyingType value ) const { return my_value/
↪value ; }
  private :
    UnderlyingType my_value ;
 } ;
```

[2]:
```
using Meter = StrongTypedef<double,struct MeterTag> ;
const double M = 1.0 ;
const double KM = 1000.0 ;
const double MM = 0.001 ;
```

[3]:
```
using Liter = StrongTypedef<double,struct LiterTag> ;
const double L = 1.0 ;
const double HL = 100.0 ;
const double CL = 0.01 ;
```

[4]:
```
#include <iostream>
```

[7]:
```
Liter fuel { 2.38*L } ;
Meter distance { 28.3*KM } ;
std::cout << (fuel/(distance/(100.*KM))) << " l/100km" << std::endl ;
```

```
8.40989 l/100km
```

Unhappily, consistency between units and multiples is not checked by the compiler : one can use a constant `L` in a `Meter` construction. Try to swap `L` and `KM` in the previous example : it does compile without any complain.

Note : in the above example, we can notice that every quantity can be divided by a quantity of a different flavor, but we do not know what should be the type of the result. As a temporary fallback, we return a value of the underlying type.

Note : in the above example, we chose arbitrarily to store the value in meters and liters. If you uses quantities which are mainly kind of liter per 100km, it is wiser to store the distances in "hundreds of kilometers". **the closer the stored values to 1, the better the precision of floating point computation**. This does not make a difference here, but may prove crucial if you handle very small or very large quantities.

1.3 Modern constants

Nowadays, we can use strong types for constant, and preprocess more stuff at compile time.

[8]:
```
template< typename UnderlyingType, typename TagType >
class StrongTypedef
 {
  public :
    constexpr explicit StrongTypedef( UnderlyingType value ) : my_value{value}␣
 ↪{}
    constexpr explicit operator UnderlyingType() const { return my_value ; }
    constexpr UnderlyingType operator*( UnderlyingType value ) const { return␣
 ↪my_value*value ; }
    constexpr UnderlyingType operator/( UnderlyingType value ) const { return␣
 ↪my_value/value ; }
  private :
    UnderlyingType my_value ;
 } ;
```

[9]:
```
template< typename UT, typename TT >
constexpr auto operator*( UT lhs, StrongTypedef<UT,TT> rhs )
 { return StrongTypedef<UT,TT>(lhs*static_cast<UT>(rhs)) ; }
```

[10]:
```
template< typename UT, typename TT1, typename TT2 >
constexpr auto operator/( const StrongTypedef<UT,TT1> & lhs, const␣
 ↪StrongTypedef<UT,TT2> & rhs )
 { return (static_cast<UT>(lhs)/static_cast<UT>(rhs)) ; }
```

[11]:
```
using Meter = StrongTypedef<double,struct MeterTag> ;
constexpr Meter M { 1.0 } ;
constexpr Meter KM { 1000.0 } ;
constexpr Meter MM { 0.001 } ;
```

[12]:
```
using Liter = StrongTypedef<double,struct LiterTag> ;
constexpr Liter L { 1.0 } ;
constexpr Liter HL { 100.0 } ;
constexpr Liter CL { 0.01 } ;
```

[13]:
```
#include <iostream>
```

[14]:
```
constexpr Liter fuel { 2.38*L } ;
constexpr Meter distance { 28.3*KM } ;
constexpr double consumption = fuel/(distance/(100.*KM)) ;
std::cout<<consumption<<" l/100km"<<std::endl;
```

```
8.40989 l/100km
```

This approach still suffers from some limitations. In particular, if you forget the parentheses around `100.*KM`, you end up multiplying by `KM` instead of dividing.

1.4 User-defined litterals

Since a long time, C++ uses suffixes to modify the type of some literals. For example, `42ll` is a `long long` type, and `3.14f` is a `float`.

C++11 introduce the possibility for developpers to define customized suffixes in the form of `_something`.

Details of this mechanism are bloody complicated (IMHO). Yet, basically, for numbers, one has to overload two flavors of the operator `operator""_something` : * one with a `long double` argument, so to deal with any kind of floating point number, * one with an `unsigned long long` argument, so to deal with any kind of integer.

For examples, we define below a `_K` suffix, which is able to multiply by 1000 any integers or floating point number :

[15]:
```
%%file tmp.quantities.cpp

#include <iostream>

long double operator ""_K ( long double value )
 { return 1000.l*value ; }

unsigned long long operator ""_K ( unsigned long long value )
 { return 1000ll*value ; }

int main()
 {
  std::cout<<5_K<<" euros"<<std::endl ;
  std::cout<<28.3_K<<" meters"<<std::endl ;
 }
```

```
Writing tmp.quantities.cpp
```

[16]:
```
!rm -f tmp.quantities.exe && g++ -std=c++17 tmp.quantities.cpp -o tmp.
 ↪quantities.exe
```

[17]:
```
!./tmp.quantities.exe
```

```
5000 euros
28300 meters
```

In the above functions, we use the same type as input and output, but actually **the returned type is at your convenience**. Usually, we will return strong types. In particular, in our first example, it is easy to replace constants by user-defined litterals which return `Meter` and `Liter` values.

BEWARE: the xeus-cling notebook kernel is not able to parse the user-defined litterals. This is why all the related examples are done with `%%file`.

1.5 In the standard library

The _ is required only for suffixes defined by developers, in order to differentiate them from the suffixes defined by the standard library.

Let's list existing suffixes below. They are always defined in a specific namespace, and **their usage requires a previous a call to using**.

For complex numbers, since C++14 : * require `using namespace std::literals::complex_literals` ; * `""if` : pur imaginary based on `float` ; * `""i` : pur imaginary based on `double` ; * `""il` : pur imaginary based on `long double`.

For durations : * require `using namespace std::literals::chrono_literals` ; * since C++14 : `""h`, `""min`, `""s`, `""ms`, `""us` et `""ns` ; * since C++20 : `""y` et `""d`.

For strings of characters : * require `using namespace std::literals::string_literals` ; * since C++14 : `""s` (`basic_string`) ; * since C++17 : `""sv` (`string_view`).

2 Questions ?

3 Exercise

Below is a copy of the example given at the beginning of this notebook. 1. Replace the constants with user-defined litterals. 2. Try to use 100km as a reference internal unit.

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

#include <iostream>

template< typename UnderlyingType, typename TagType >
```

```
class StrongTypedef
 {
  public  :
    constexpr explicit StrongTypedef( UnderlyingType value ) : my_value{value}␣
  ↪{}
    constexpr explicit operator UnderlyingType() const { return my_value ; }
    constexpr UnderlyingType operator*( UnderlyingType value ) const { return␣
  ↪my_value*value ; }
    constexpr UnderlyingType operator/( UnderlyingType value ) const { return␣
  ↪my_value/value ; }
  private :
    UnderlyingType my_value ;
 } ;

template< typename UT, typename TT >
constexpr auto operator*( UT lhs, StrongTypedef<UT,TT> rhs )
 { return StrongTypedef<UT,TT>{lhs*static_cast<UT>(rhs)} ; }

template< typename UT, typename TT1, typename TT2 >
constexpr auto operator/( const StrongTypedef<UT,TT1> & lhs, const␣
  ↪StrongTypedef<UT,TT2> & rhs )
 { return (static_cast<UT>(lhs)/static_cast<UT>(rhs)) ; }

template< typename UT, typename TT >
auto & operator<<( std::ostream & os, StrongTypedef<UT,TT> data )
 { return (os<<static_cast<UT>(data)) ; }

using Liter = StrongTypedef<double,struct LiterTag> ;

// TO BE COMPLETED

using Meter = StrongTypedef<double,struct MeterTag> ;

// TO BE COMPLETED

int main()
 {
  constexpr auto fuel { 2.38_L } ;
  constexpr auto distance { 28.3_KM } ;
  constexpr auto consumption = fuel/(distance/100._KM) ;
  std::cout<<consumption<<" l/100km"<<std::endl;
 }
```

```
Overwriting tmp.quantities.cpp
```

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

```
tmp.quantities.cpp: In function 'int main()':
tmp.quantities.cpp:36:25: error: unable to find numeric literal operator
'operator""_L'
   36 |   constexpr auto fuel { 2.38_L } ;
      |                         ^~~~~~
tmp.quantities.cpp:36:25: note: use '-fext-numeric-literals' to enable more
built-in suffixes
tmp.quantities.cpp:37:29: error: unable to find numeric literal operator
'operator""_KM'
   37 |   constexpr auto distance { 28.3_KM } ;
      |                             ^~~~~~~
tmp.quantities.cpp:37:29: note: use '-fext-numeric-literals' to enable more
built-in suffixes
tmp.quantities.cpp:38:47: error: unable to find numeric literal operator
'operator""_KM'
   38 |   constexpr auto consumption = fuel/(distance/100._KM) ;
      |                                               ^~~~~~~
tmp.quantities.cpp:38:47: note: use '-fext-numeric-literals' to enable more
built-in suffixes
```

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

```
sh: 1: ./tmp.quantities.exe: not found
```

3.1 Resources & inspirations

- https://akrzemi1.wordpress.com/2012/08/12/user-defined-literals-part-i/
- https://en.cppreference.com/w/cpp/language/user_literal