1-units

June 3, 2024

1 Strong typing and physical units

1.1 Motivation

There are many situations where the straight usage of builtin types is not safe.

For example, we could wish to create a circle from a radiss or a diameter :

[28]:
```
class Circle
 {
  public:
    explicit Circle(double radius)
     : my_radius(radius) {}
    explicit Circle(double diameter)
     : my_radius(diameter / 2) {}
  private:
    double my_radius ;
 } ;
```

```
input_line_35:6:14: error: constructor cannot be

redeclared
    explicit Circle(double diameter) // DOES NOT COMPILE
             ^

input_line_35:4:14: note: previous definition is
here
    explicit Circle(double radius)
             ^
```

```
Interpreter Error:
```

Other example : let's take a rectangle constructor from one width and height. In the client code, there is no way to distinguish which is width and which is height, and it is (so) easy to swap these values.

[11]:
```
class Rectangle
 {
  public:
    Rectangle( double width, double height ) { /*...*/ ; }
    //....
 } ;
```

[12]:
```
//...
Rectangle r(10, 12) ;
```

In both cases (and in many others), we may wish to **create specific types to distinguish these quantities**.

Type aliasing enables to better express the intention, and improve readability.

[13]:
```
using Width = double ;
using Height = double ;

class Rectangle
 {
  public:
    Rectangle( Width w, Height h ) { /*...*/ ; }
    //....
 } ;
//...
Rectangle r(Width{10},Height{12}) ;
```

But from the compiler point of view, **it is still one single type**, and any erroneous mixture is allowed.

We need new types which are considered different and not commutable by the compiler. The C++ community calls this possible feature **strong typedef** or **opaque typedef**. It was proposed to the normalization committee of C++, but it seems Bjarne Stroustrup is not in favor of it…

1.2 Hand-made *strong typedef*

We want, for example, to create a new type which behaves exactly like a `double`, but will be considered as a **different type** by the compiler.

First steps consist in creating a constructor and a conversion operator from and to `double`. **Let's make them `explicit`**, so to avoid any unwanted implicit conversion:

[1]:
```
class Width
 {
  public :
    explicit Width( double value ) : my_value{value} {}
    explicit operator double() const { return my_value ; }
  private :
    double my_value ;
 } ;
```

Unfortunately, we cannot do much with this type. Firstly, **all usual operators are missing**, and you can guess how it will be painful to redo all this work each time we want to add a new "strong alias".

1.3 Let's use a template

To ease the writing of strong alias, we can obviously use templates.

This one, beyond constructor and converter, add the `+` operator and `<<` operator:

[16]:
```
template< typename UnderlyingType >
class StrongTypedef
 {
  public :
    explicit StrongTypedef( UnderlyingType value ) : my_value{value} {}
    explicit operator UnderlyingType() const { return my_value ; }
    friend StrongTypedef operator+( StrongTypedef lhs, StrongTypedef rhs )
     { return StrongTypedef(lhs.my_value+rhs.my_value) ; }
    friend std::ostream & operator<<( std::ostream & os, StrongTypedef v )
     { return (os<<v.my_value) ; }
  private :
    UnderlyingType my_value ;
 } ;

using Width = StrongTypedef<double> ;
using Height = StrongTypedef<double> ;
```

Well, from the compiler point of view, `Width` and `Height` are again the same single type `StrongTypedef<double>`. Let's add a second type parameter, whose single utility is to make them somehow different and not commutable.

[17]:
```
template< typename UnderlyingType, typename TagType >
class StrongTypedef
 {
  public :
    explicit StrongTypedef( UnderlyingType value ) : my_value{value} {}
    explicit operator UnderlyingType() const { return my_value ; }
    friend StrongTypedef operator+( StrongTypedef v1, StrongTypedef v2 )
     { return StrongTypedef(v1.my_value+v2.my_value) ; }
    friend std::ostream & operator<<( std::ostream & os, StrongTypedef v )
     { return (os<<v.my_value) ; }
  private :
    UnderlyingType my_value ;
 } ;

struct WidthTag {} ;
struct HeightTag {} ;

using Width = StrongTypedef<double,WidthTag> ;
using Height = StrongTypedef<double,HeightTag> ;
```

Void structs `WidthTag` and `HeightTag` aims to differenciate between `Width` et `Height`. There are usually called **Tag Types** or **Phantom Types**. We can even create them on the fly :

[18]:
```
using Width = StrongTypedef<double,struct WidthTag> ;
using Height = StrongTypedef<double,struct HeightTag> ;
```

1.4 Factorize and recombine operators

Actually, not all operators are valid for all types. For example, if we mix both points and vectors, it makes sense to add vectors, but not to add points… A single template which would define all operators is not relevant. We must find a mechanism so to **add operators on-demand**, depending of the meaning of the new type.

[5]:
```
template< typename UnderlyingType, typename TagType >
class StrongTypedef
 {
  public :
    explicit StrongTypedef( UnderlyingType value ) : my_value{value} {}
    explicit operator UnderlyingType() const { return my_value ; }
  private :
    UnderlyingType my_value ;
 } ;
```

[6]:
```
#include <iostream>
template< typename UT, typename TT >
std::ostream & operator<<( std::ostream & os, const StrongTypedef<UT,TT> & obj )
 { return (os<<static_cast<UT>(obj)) ; }
```

[7]:
```
template< typename UT, typename TT >
struct addition
 {
  friend StrongTypedef<UT,TT> operator+( const StrongTypedef<UT,TT> & lhs,␣
 ↪const StrongTypedef<UT,TT> & rhs )
   { return StrongTypedef<UT,TT>(static_cast<UT>(lhs)+static_cast<UT>(rhs)) ; }
 } ;
```

[8]:
```
struct Meter : StrongTypedef<double, Meter>, addition<double,Meter>
 { using StrongTypedef::StrongTypedef ; } ;
```

[9]:
```
struct Second : StrongTypedef<double, Second>, addition<double,Second>
 { using StrongTypedef::StrongTypedef ; } ;
```

[10]:
```
Meter m { 1000 } ;
Second s { 60 } ;

std::cout<<m<<"+"<<m<<"="<<(m+m)<<std::endl ;
std::cout<<s<<"+"<<s<<"="<<(s+s)<<std::endl ;
std::cout<<m<<"+"<<s<<"="<<(m+s)<<std::endl ;
```

```
input_line_17:7:30: error: invalid operands to binary
expression ('__cling_N59::Meter' and '__cling_N510::Second')
std::cout<<m<<"+"<<s<<"="<<(m+s)<<std::endl ;
                            ~^~

input_line_14:4:31: note: candidate function not
viable: no known conversion from '__cling_N510::Second' to 'const
StrongTypedef<double, __cling_N59::Meter>' for 2nd argument
  friend StrongTypedef<UT,TT> operator+( const StrongTypedef<UT,TT> & lhs, const
StrongTypedef<UT,TT> & rhs )
                              ^

input_line_14:4:31: note: candidate function not
viable: no known conversion from '__cling_N59::Meter' to 'const
StrongTypedef<double, __cling_N510::Second>' for 1st argument
```

```
Interpreter Error:
```

1.5 Libraries to the rescue

Strong typing gives a meaning to your numerical values, and therefore helps the compiler to detect inconsistencies. But as you can guess, writing a complete "strong typedef" is rather complex. Some experts provide turnkey libraries : * type_safe, by Jonathan Muller * NamedType, by Jonathan Boccara * opaque-typedef, by Kyle Markley * BOOST_STRONG_TYPEDEF

2 Questions ?

2.1 Exercise : give units to variables

One obvious application of strong typedefs is to associate physical units to variables. In the code below, you are asked to introduce the types `Km` and `Liter`.

[11]:
```
%%file tmp.units.cpp

#include <iostream>

// TO BE COMPLETED

class Journey
 {
  public :
    Journey( double distance, double fuel )
     : my_distance{distance}, my_fuel{fuel} {}
    double consumption() { return my_fuel/my_distance*100. ; }
  private :
    double my_distance ;
    double my_fuel ;
} ;

int main()
 {
  double distance { 28.3 } ;
  double fuel { 2.38 } ;
  Journey j { fuel, distance } ;
  std::cout<<j.consumption()<<" l/100km"<<std::endl;
 }
```

```
Writing tmp.units.cpp
```

[12]:
```
!rm -f tmp.units.exe && g++ -std=c++17 tmp.units.cpp -o tmp.units.exe
```

[13]:
```
!./tmp.units.exe
```

```
1189.08 l/100km
```

2.2 Ressources & inspirations on strong typing

Blogs & Tutrials * Jonathan Muller : https://foonathan.net/2016/10/strong-typedefs/ * Jonathan Boccara : https://www.fluentcpp.com/2016/12/08/strong-types-for-strong-interfaces/ * Kyle Markley, 2016 : https://sourceforge.net/p/opaque-typedef/wiki/Home/

Libraries * Jonathan Muller : https://github.com/foonathan/type_safe * Jonathan Boccara : https://github.com/joboccara/NamedType * Boost : http://www.boost.org/doc/libs/1_61_0/libs/serialization/doc/strong_typedef.html * Kyle Markley : https://sourceforge.net/projects/opaque-typedef/files/

© CNRS 2020
Assembled and written by David Chamont, translated by Pierre Aubert, this work is made available according to the terms of the
Creative Commons License - Attribution - NonCommercial - ShareAlike 4.0 International