Computer Science

Units in C++

ACCU 2016

slides: http://wiki.hsr.ch/PeterSommerlad/

IFS INSTITUTE FOR SOFTWARE

Prof. Peter Sommerlad
Director IFS Institute for Software
Bristol April 2016

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Problem with Primitives

- Typical beginners problem and solution.
- can you spot the errors?
- more problems?

```
#include <iostream>

int main(){
  using std::cout;
  using std::endl;
  using std::cin;
  cout << "Enter km driven:"<< endl;
  double x{},y{};
  cin >> x;
  cout << "Enter liters:" << endl;
  cin >> y;
  cout << "You used " << x/y << " liters per km\n";
}
```

double considered harmful

float as well

Representing Values with a dimension

- In all OO languages it can make sense to model quantities as types beyond the primitive numerical types
  - and in C++ such an abstraction can be overhead free!
- Type system can help to avoid strange mistakes, such as dividing apples by oranges
  - This is really why we want Units!
- See also Value Object Patterns (by Kevlin Henney), esp. “Whole Value” pattern.

Whole Value Pattern

what does a number alone stand for?

Whole Value Pattern (Kevlin Henney)

- How can you represent primitives quantities from your problem domain without loss of meaning?
  - integers and floating point numbers are not very useful!
  - lack of dimension, intent communication, no compile time checking
- Express the quantity as a class.
  - a Whole Value recovers the loss of meaning and checking by providing a Dimension and Range.
  - can wrap simple types (or attribute sets)

A better version?

- At least division makes sense only in one direction or the output will show.
- However, this requires a lot of scaffolding.

```
int main(){
  using namespace std;
  cout << "Enter km driven:"<< endl;
  units::volume x{};
  units::distance y{};
  cin >> y;
  cout << "Enter liters:" << endl;
  cin >> x;
  cout << "You used " << x/y << "\n";
  //cout << "You used " << y/x << “\n"; // error
}
```

With C++11/14 you can write:

- UDLs
- constexpr
- Apply Whole Value Pattern
  - distance in km
  - volume in liters
  - usage in l/100km
- Overload Operators
  - arithmetic, output

```
int main(){
  using namespace std;
  using namespace units::literals;
  cout << "This program will show your petrol usage\n"
     << "For example, if you enter " << 500_km << endl
     << "and " << 40_l << endl
     << "you should receive " << (40_l/500_km) << endl;
//. . .
```

```
This program will show your petrol usage
For example, if you enter 500 km
and 40 l
you should receive 8 l/100km
```

DIY simple units

- Many things to do…
- Wrap simple value with class
- Provide explicit conversion constructor
- Provide useful arithmetic
  - multiplication with factors
  - addition with same unit
  - comparison/conversion/IO/…

```
namespace units{
struct distance
     :private boost::multiplicative<distance,double>
     ,boost::addable<distance>{
   explicit constexpr distance(double km=0):in_km{km}{};
   constexpr distance operator *=(double km){
      in_km*=km;
      return *this;
   }
   constexpr distance operator/=(double d){
      return *this *= 1/d;
   }
   constexpr distance operator +=(distance const &other){
      in_km+=other.in_km;
      return *this;
   }
   constexpr distance operator -=(distance const &other){
      return *this += distance{-other.in_km};
   }
   friend std::ostream & operator<<(std::ostream &out,distance const &v){
      return out << v.in_km << " km ";
   }
   friend std::istream & operator>>(std::istream &in,distance  &v){
      return in >> v.in_km ;
   }
   double in_km;
};
```

DIY Simple Units: UDL Literals

- C++11/14 allow defining suffix operators
- For numerical values you will always have to provide 2 of them, one for integers and one for floating point constants
- NB: Often the integral one will need to return the value converted to a floating point
  - Was a bug in PhysUnitCPP11

```
namespace literals{
constexpr auto operator""
_km(unsigned long long d){
  return distance(d);
}
constexpr auto operator""
_km(long double d){
  return distance(d);
}
}
```

DIY Simple Units: Mixed Arithmetic

- Can be tricky/burdensome to get right
- beware of automatic conversions
  - conversion ctors and operators always `explicit`
- define the base unit consistently, meters vs kilometers vs miles, liters vs. gallons
- might require `friends`
  - beware of accessors
  - use `{value}` for construction to avoid narrowing conversions

```
struct usage{
  explicit constexpr usage(double l_100km=0)
   :liters_100km{l_100km}{};
  friend std::ostream &
   operator<<(std::ostream &out,usage const &v){
    return out << v.liters_100km << " l/100km ";
  }
private:
  double liters_100km;
};
struct volume {
  explicit constexpr volume(double liters=0)
   :in_liters{liters}{}
...
};
usage operator/(volume const &v,distance const &d){
    return usage{100*v.in_liters/d.in_km};
}
```

© Peter Sommerlad

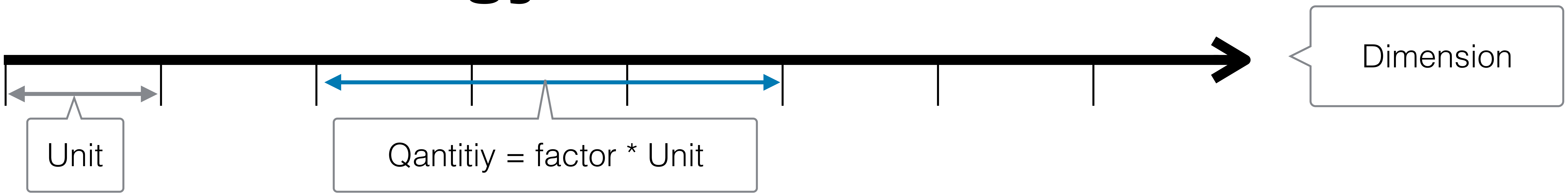
DIY Units Summary

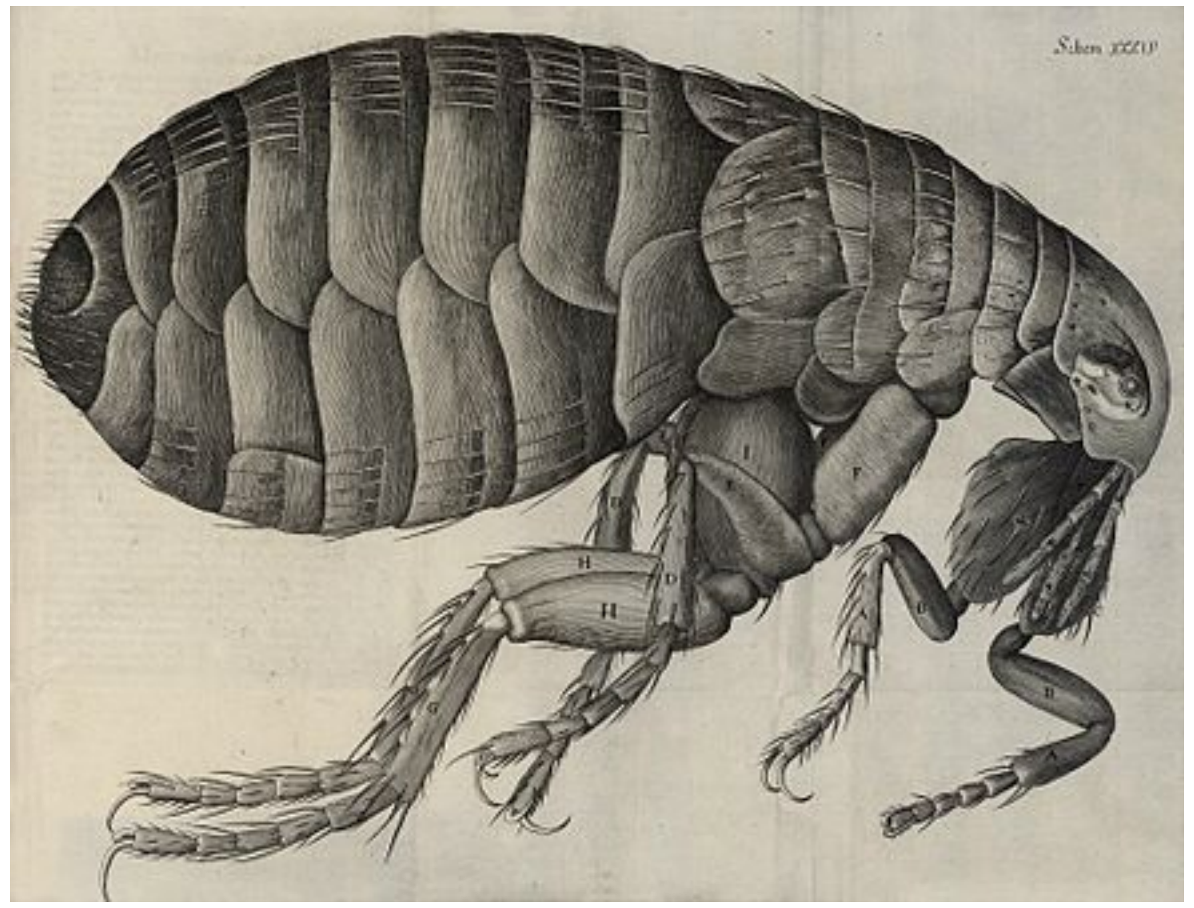
- Can be worth it! regular numbers only represent factors not quantities.
- Requires conscious design and thinking
- Using constexpr can reduce run-time computation
- Focus on usability not features in your application domain
- Stick to metrical units (IMHO) or those that makes sense…
- Stay to a single internal base unit, i.e., meters for distances and provide inward/outward conversions, e.g., with UDL operators

Humans and Units

- My observation:
  - humans tend to stick to units that allow small numbers 1-100*n to represent useful quantities
  - often only 2-3 digits practically relevant
  - except for trained engineers, 1e10 can be hard to grasp
- Guideline:
  - if you DIY units, consider those observations and provide appropriate conversions and UDL suffixes for your domain (_km,_mm)

Terminology of Units Libraries





- Dimension (distance)
- Unit (flea hop [fh], cubit)
- Quantity (7 fh, 2 cubits)
- System (SI): set of (derived) units for a set of dimensions
- Problem: easy to mix up terminology for inexperienced users
- Problem: what to put into the (C++) type system?

SI Dimensions and Base Units

	Dimension	Symbol(s)	Base Unit	Unit Symbol	Synonyms
1	length	$l,x,y,r,\ldots$	Meter	m	distance
2	mass	m	Kilogramm	kg	“weight"
3	time	t	Second	s	duration, time interval
4	electric current	I,i	Ampere	A	
5	thermodynamic temperature	T	Kelvin	K	“heat”, “temperature”
6	amount of substance	n	Mole	mol	
7	luminous intensity	I_v	Candela	cd	

SI derived Units (excerpt)

Quantity	Unit Name	Symbol(s)	as SI Unit	as Base Units
frequency	*Hertz*	*Hz*		s^{-1}
force	*Newton*	*N*		$kg\ m\ s^{-2}$
energy	*Joule*	*J*	Nm	$kg\ m^2\ s^{-2}$
power	*Watt*	*W*	J/s	$kg\ m^2\ s^{-3}$
electric potential difference, voltage, electric tension	*Volt*	*V*	W/A	$m^2kgs^{-3}A^{-1}$
electric resistance	*Ohm*	*Ω*	V/A	$m^2kgs^{-3}A^{-2}$

SI prefixes (ratios)

- like kilo, milli, mega…
  - sometimes as factors
  - sometimes as factors of a specific dimensionless unit type
- std::ratio defines those also as compile-time ratios
  - as types representing a quotient
  - no direct conversion to long double
- beware of name clashes

```
// PhysUnit-CT-Cpp11
constexpr long double yotta = 1e+24L;
constexpr long double zetta = 1e+21L;
constexpr long double   exa = 1e+18L;
constexpr long double  peta = 1e+15L;
constexpr long double  tera = 1e+12L;
constexpr long double  giga = 1e+9L;
constexpr long double  mega = 1e+6L;
constexpr long double  kilo = 1e+3L;
constexpr long double hecto = 1e+2L;
constexpr long double  deka = 1e+1L;
constexpr long double  deci = 1e-1L;
constexpr long double centi = 1e-2L;
constexpr long double milli = 1e-3L;
constexpr long double micro = 1e-6L;
constexpr long double  nano = 1e-9L;
constexpr long double  pico = 1e-12L;
constexpr long double femto = 1e-15L;
constexpr long double  atto = 1e-18L;
constexpr long double zepto = 1e-21L;
constexpr long double yocto = 1e-24L;
// <ratio>
typedef ratio<1LL, 1000000000000000000LL> atto;
typedef ratio<1LL,    1000000000000000LL> femto;
typedef ratio<1LL,       1000000000000LL> pico;
typedef ratio<1LL,          1000000000LL> nano;
typedef ratio<1LL,             1000000LL> micro;
typedef ratio<1LL,                1000LL> milli;
typedef ratio<1LL,                 100LL> centi;
typedef ratio<1LL,                  10LL> deci;
typedef ratio<                 10LL, 1LL> deca;
typedef ratio<                100LL, 1LL> hecto;
typedef ratio<               1000LL, 1LL> kilo;
typedef ratio<            1000000LL, 1LL> mega;
typedef ratio<         1000000000LL, 1LL> giga;
typedef ratio<      1000000000000LL, 1LL> tera;
typedef ratio<   1000000000000000LL, 1LL> peta;
typedef ratio<1000000000000000000LL, 1LL> exa;
```

Boost Units

```
#include "boost/units/quantity.hpp"
#include "boost/units/systems/si.hpp"
#include "boost/units/systems/si/prefixes.hpp"
#include "boost/units/io.hpp"
```

- pre-C++11 design
- heavily relies on macros and Boost::mpl
- hyper-flexible design: you can define your own units system, e.g. based on “flea hops”
- compile-time checking (with interesting error messages)
- designed by experts for experts (in C++ and unit systems)
- several Systems provided (SI, CGS, trigonometry, temperature, …)
- Base Units from Astronomical to US Units

Boost Units example

- Need to include the right headers
- requires using namespace
  - note: Cevelop can refactor that!
- Mistakes make interesting compile errors
  - almost impossible to see, what was wrong
  - requires trial and error
  - e.g., using integers for factors!

```
#include <iostream>
#include "boost/units/quantity.hpp"
#include "boost/units/systems/si.hpp"
#include "boost/units/systems/si/prefixes.hpp"
#include "boost/units/io.hpp"

int main(){
   // don't: using namespace std;
   // there are kilo/milli etc also in namespace std! <ratio>
   using namespace boost::units;
   using namespace boost::units::si;
   std::cout << "Enter km driven:" << std::endl;
   double y;
   std::cin >> y;
   quantity<length> dist{ y * kilo * meters };
   std::cout << "you drove " << engineering_prefix << dist;
   std::cout << "\nEnter liters:" << std::endl;
   double x;
   std::cin >> x;
   quantity<volume> const liter { 1 * milli * cubic_meter};
   // could define unit liter instead
   quantity<volume> vol= x * liter;
   std::cout << "You used " << 100 * x / y << " l/100km\n";
   std::cout << "You used " << 1e2 * vol / dist << "\n";
}
```

PhysUnit-CT-Cpp11

```
#include <phys/units/quantity.hpp>
#include <phys/units/io.hpp>
```

- C++11 design
- Employs C++11 features
- With auto even easier to use
- requires using namespace for output and UDL operators
- “units” modeled as constants of corresponding quantities with a factor

- Representation is double unless redefined with a Macro :-(
- “Only” SI system (7 dimensions)
  - but with dimensional analysis
  - no “dimensionless” quantities (just numbers then)

Dimesional Analysis - what?

- Energy = Force * Distance

  $$\mathrm{kg\ m^2\ s^{-2}} = \mathrm{kg\ m\ s^{-2}} * \mathrm{m}$$

- Power = Energy / Time

  $$\mathrm{kg\ m^2\ s^{-3}} = \mathrm{kg\ m^2\ s^{-2}} / \mathrm{s}$$

- $E = mc^2$

- Adjust the exponents per dimension accordingly when multiplying quantities

```
template< int D1, int D2, int D3, int D4 = 0, int D5 = 0, int D6 = 0, int D7 = 0 >
struct dimensions
{
    enum
    {
        dim1 = D1,
        dim2 = D2,
        dim3 = D3,
        dim4 = D4,
        dim5 = D5,
        dim6 = D6,
        dim7 = D7,
        is_all_zero =
            D1 == 0 && D2 == 0 && D3 == 0 && D4 == 0 && D5 == 0 && D6 == 0 && D7 == 0,
        is_base = 1 == (D1 != 0) + (D2 != 0) + (D3 != 0)
                  + (D4 != 0) + (D5 != 0) + (D6 != 0) + (D7 != 0)
            &&
            1 ==  D1 + D2 + D3 + D4 + D5 + D6 + D7,
    };

    template< int R1, int R2, int R3, int R4, int R5, int R6, int R7 >
    constexpr bool operator==( dimensions<R1, R2, R3, R4, R5, R6, R7> const & ) const
    {
        return D1==R1 && D2==R2 && D3==R3 && D4==R4 && D5==R5 && D6==R6 && D7==R7;
    }

    template< int R1, int R2, int R3, int R4, int R5, int R6, int R7 >
    constexpr bool operator!=( dimensions<R1, R2, R3, R4, R5, R6, R7> const & rhs )
const
    {
        return !( *this == rhs );
    }
};
```

PhysUnits-CT-Cpp11 Example

- Required patches for C++14
  - const for constexpr member-functions
- a bit simpler to use
- still beware of floating point vs. integers (patch for UDL operators just applied)
- SI base dimensions

```
#include <iostream>

#include <phys/units/quantity.hpp>
#include <phys/units/io.hpp>
int main(){
   using std::cout;
   using std::endl;
   using std::cin;
   using namespace phys::units; // types
   using namespace phys::units::literals; // UDL
   using namespace phys::units::io::eng; // <<(ostream&,...)
   cout << "Enter km driven:"<< endl;
   double x{};
   cin >> x;
   quantity<length_d> dist = x * kilo * meter;
   cout << "Enter liters:" << endl;
   double y{};
   cin >> y;
   quantity<volume_d> vol = y * liter;
   cout << "You used " << 100*y/x << " liters per 100 km\n";
   auto res = vol*100_km/dist;
   cout << "You used " << res << "\n";
}
```

Boost::Units vs. PhysUnit-CT-Cpp11

Dimension	Boost dimension	Boost Unit Type, Unit	PhysUnit Dimension	PhysUnit Unit
length	length_dimension	length, meter	length_d	meter
mass	mass_dimension	mass, kilogramm	mass_d	kilogramm
time	time_dimension	time, second	time_interval_d	second
electric current	current_dimension	current, ampere	electric_current_d	ampere
thermodynamic temperature	temperature_dimension	temperature, kelvin	thermodynamic_temperature_d	kelvin
amount of substance	amount_dimension	amount, mole	amount_of_substance_d	mole
luminous intensity	luminous_intensity_dimension	luminous_intensity, candela	luminous_intensity_d	candela

Switching between unit libraries?

- Beware of synonyms
  - Boost Units often defines different spellings and singular and plural for unit values (meter, metre, meters, metres)
  - Names for dimensions and base units can differ
- different programming models
  - Boost Units packs a lot of information into types and templates
  - PhysUnit-CT-Cpp11 only provides quantities and 7 SI dimensions

Are they performant?

```
template< typename T1, typename T2, typename T3 >
T1 do_work( T1 v1, T2 v2, T3 v3 )
{
    // Do a bunch of work.  We don't really care about the answer;
    // this is just to exercise addition, subtraction, multiplication, and division.
    T1 x1 = v1;
    T2 x2 = v2;
    T3 x3 = v3;
    for( int i = 0; i < meg; i++ )
    {
        for( int j = 0; j < k; j++ )
        {
            x2 = -x2 - v2;
            x3 *= 1.00002;
            x1 += x2 / x3;
        }
    }
    return x1;
}
//////////
    high_resolution_clock clock{};
    auto const t0 = clock.now();
    // do some work with doubles;
    volatile double d{};
    for (int i = 1; i < 11; ++i){
       d += do_work(i * 0.1, 0.2, 0.3);
    }
    auto const t1 = clock.now();
    // do exact same work with quantity
    //quantity< velocity_dimension >
    quantity<velocity> s{};
    for (int i = 1; i < 11; ++i){
       s += do_work(i * 0.1*meter/second, 0.2*meter, 0.3*second);
    }

    auto const t2 = clock.now();
```

```
Peter-Sommerlads-Dienstlich-mbpro:units_ws sop$ Time/Release/
Time
Time/Release/Time: Performance test of quantity library.
one double work loop     = 2068447 usec  (1)
one quantity work loop   = 2048642 usec  (0.99)
d = -1.6667e+05
s = -1.6667e+05 m s-1

Peter-Sommerlads-Dienstlich-mbpro:units_ws sop$ Time_boost_unit/
Release/Time_boost_unit
Time_boost_unit/Release/Time_boost_unit: Performance test of
quantity library.
one double work loop     = 2089602 usec  (1)
one quantity work loop   = 2057997 usec  (0.985)
d = -1.6667e+05
s = -1.6667e+05 m s^-1
```

Will we get Units in the C++ standard?

- You already have…

  one

  std::chrono::seconds

- Is the design appropriate?
- Wrong question!
- What are the trade-offs of std::chrono's design

Dimension std::chrono::duration

- allows for integral representation types (count())
- encodes scaling into type as non-type template argument
- Using floating point representation can suffer
- base unit not available as factor
- 1s and 1.s are different types
- duration_cast<D>(d) required

```
//                    Represenation, Scale as std::ratio
typedef duration<long long,            nano> nanoseconds;
typedef duration<long long,           micro> microseconds;
typedef duration<long long,           milli> milliseconds;
typedef duration<long long                 > seconds;
typedef duration<     long, ratio<  60> > minutes;
typedef duration<     long, ratio<3600> > hours;

constexpr chrono::seconds operator"" s(unsigned long long __s)
{
    return chrono::seconds(static_cast<chrono::seconds::rep>(__s));
}

constexpr chrono::duration<long double> operator"" s(long double __s)
{
    return chrono::duration<long double> (__s);
}
```

some inspiration by
Boost::Units happened!

What about other SI units in std?

- I believe the std::chrono design is not appropriate for other units
  - for physical computations floating point rules
  - integers are not that interesting (may be except on specific hardware, like FPGA/DSP)
- There is a need: you can help specify it!
  - If I find time, I’ll give it a try, based on PhysUnit’s design, except for the macros :-)

Will a std units library work?

- Must be teachable! Boost::Unit is working hard against beginners
- Must be efficient = same as using double directly!
- Must allow human-graspable value ranges:
  - 10km is easier than 1e4m
- Must have <put your requirements here> !

Wrap up

- Apply the Whole Value pattern to your code!
- C++11/14/17 allow efficient encoding of quantities with strong typing
  - strong typing = compile errors!
- Either DIY domain-specific - catching subtle errors at compile time
- Or learn to use a units library
- Help specifying one for the standard!

Questions?

- contact: peter.sommerlad@hsr.ch

- Looking for a better IDE:

Cevelop
Your C++ code deserves it

- examples become available at: https://github.com/PeterSommerlad/Publications

Download IDE at:
www.cevelop.com