Apples and Oranges: The Lightweight UnitsC++ Library

Copyright © 2008 Don Peterson
someonesdad1@gmail.com
Updated: 20 Jun 2014

Note: this library was originally named UnitsC++ (which I'll keep in this document) and submitted to Sourceforge in 2008. However, I'd rather have it hosted on Google, so I've submitted it there under the name `unitscpp` (http://code.google.com/p/unitscpp/).

Table of Contents

Introduction

In doing numerical calculations, we all learn the importance of checking our equations for dimensional consistency. However, in using computers for numerical computations, vast numbers of bugs have been created by equating apples to oranges. The problem is not helped by programming languages that constrain all numerical quantities to a few numerical types (typically one floating point type).

An example of a very common error is when a programmer defines a `double` variable `angle` to be in degrees, as this is a common unit used in practical situations. However, should calculations need to be done with the C library trig functions, the programmer needs to remember that any call to a trig function needs to convert `angle` to radians. This is an easy thing to forget -- both by the original programmer and the maintenance programmer.

In this article I discuss a C++ library called **UnitsC++** that elevates numerical quantities with different units to different types, which keeps apples in their baskets and oranges in their crates. The goal is to show you a way to reduce numerical errors in your programs by preventing you from treating apples and oranges as the same objects – if you do, you'll get a compile-time error.

Why you might want to use UnitsC++

1. If you write C++ programs that calculate with numbers that represent quantities of objects or physical units, you should consider using this library.

2. It helps make numerical code easy to read and more likely to be written correctly.
3. It's lightweight: small, little runtime cost in space or time, and you don't need to be a computer scientist to understand it.
4. It's simple. Create and use only what you need.
5. The library adapts to your problem domain, not the other way around.
6. It's implemented in one include file generated by a python script from a configuration file. Units, derived units, and constants are easily added or removed in the configuration file. You should never need to edit the include file.
7. The configuration file has a simple syntax. You can define derived units in a natural way by expressions using other fundamental and derived units. Multiplication, division, and integer exponentiation are supported.
8. It should be portable and usable with many C++ compilers. The compiler primarily just needs to support templates and the addition of integer template parameters. I developed and tested it with the MinGW 3.4.2 g++ compiler, the cygwin 3.4.4 g++ compiler, and the Borland 5.5.1 compiler on Windows. In June of 2014, I tested it with python 2.7.6 and 3.4.0 on a Linux box using the GNU g++ 4.8.2 compiler.
9. A set of SI units, derived units, and constants are provided by using the python script with the `-s` option. While these won't be complete for many users, it's easy to customize the list for your own uses by editing the python script.

For help in converting amongst a variety of units, I recommend the GNU units program [GNUunits]. I find this program indispensable.

Some things UnitsC++ does not do

1. Support fractional exponents.
2. Serialize unit objects to and from byte stream representations.

If you want fractional exponents, it might not be too hard to change the `unitscpp.py` script to define unit dimensions in units of some fraction. For example, the integer values for Length could be in "units" of 1/360, allowing numerous square and cube roots to be taken. You'd have to write the extra template code for the root functions.

Background

Barton and Nackman's insight [BN1994] into using C++ templates to provide types for units is a useful technique and is the basis for what I discuss here. Brown [Brown2001] implemented the SIunits library using similar ideas, as did Kenniston [Kenn2002]. The mcs::units project [SFmcs::units] was apparently once part of Boost, but doesn't currently appear on Boost's pages anymore. These are examples of "heavyweight" implementations.

After examining the heavyweights, it struck me that they might have worked too hard to solve the problem. In particular, they use template metaprogramming techniques, which can burden older compilers. The authors of the heavyweights apparently spent substantial effort on overcoming limitations in various compilers, a familiar and frustrating task when trying to write platform-independent code. mcs::units comments that their library does tend to stress compilers and that you'll likely need a current compiler. I felt there was a niche for a library that tries to keep the solution simple and not try to solve the general case.

I came across Abhishek Padmanabh's elegantly simple and clean implementation [Pad2007] that was intended to illustrate the principle of templates providing different types for various units. I felt his implementation provided most of what's needed for bread-and-butter numerical calculations, so I decided to base my implementation on it.

Important numbers = types

A few decades ago, I wrote a computer model of a manufacturing process to help make financial decisions. I spent a good fraction of my time in solving the problem on writing down the equations and making sure they were correct (both dimensionally and numerically). These equations involved "units" like dollars, hours, widgets, sputter targets, people, polishing compound, etc. This led to the insight

"Units" are numerical concepts that are important to the problem domain.

This insight is more general than e.g. using the usual seven fundamental SI units. By "elevating" a concept to a unit, the programmer is stating that this concept is important in the solution of his problem and he wants to make sure he gets it right. A unit is "orthogonal" to all other units and cannot be mixed with other units in undefined ways.

Here's an example that was mentioned in the introduction. Angle measure is not a fundamental SI unit. Mathematically, this is fine, as the radian is defined as the ratio of the length of an arc to a radius and is thus a dimensionless number. Practically, it's a disaster. We assign the `double` type to an angle variable in our code and then blithely think we can keep our work in radians and degrees straight. Our code gets littered with conversion factors, as we frequently like to work in degrees but the `cmath` library functions want arguments in radians. I don't have enough integers at hand to enumerate the times I've made trig argument mistakes. Ah, I see that gentleman in the back row nodding his head in agreement.

With the above insight, the solution is trivial: make angle measure a fundamental unit. This means you think it's important enough to ask the compiler to help you keep track of angles and their manipulations by constructing a unique type for it. It's that simple and easy. Then it's a short trudge over to the realization that you can ask for this help for all the numbers in your program should you wish to.

Static type checking is the technique the compiler uses to help us. See [Strous1997] section 24.2.3 for some thoughts on this. **For mission-critical code, I want all the type checking I can get**. If you don't agree with this, I can point to some $125 million chump change which might persuade you to rethink this [NASA1999]. I can translate the above insight into programmer-speak:

If it's important, make it a type.

While thinking about the above and iteratively implementing this units library, I decided on the following requirements for the library in roughly decreasing priority:

Characteristic	Comments
Keep it simple	Users will be scientists and engineers, not computer scientists. The library should be easy to understand and use.
Easy to add or remove a dimension	Problems always change, so the library must easily support change without changing lots of code.
Portability	A possible side effect of simplicity. Don't put too many demands on the compiler. And don't compromise the design to cater to a broken compiler.
Features	Arithmetic, ordering, and equality testing. Easy unit conversions. Talk to streams too.
Be independent	Do not use other libraries or tools unless they're part of the package. [*Technically, I failed this requirement*

Characteristic	Comments
	because of the need for python.]
No kitchen sink syndrome	Don't try to handle all cases or make the library bulletproof against all misuses.
Testability	Make it easy to test.
Documentation	Show the common use cases and the library's limitations.
Easy to understand	Try to write it so users can understand the implementation and adapt it to their needs.

Implementation

The implementation and feature set went through numerous iterations. The first iteration was based on defining the template parameters as macros, which lead to a clean and easy to read include file implementation. The textual "noise" of the template parameters was hidden and it was easy to see how simple the implementation was. But C++ programmers shouldn't use macros.

I switched to an implementation that used a python script `unitscpp.py` to generate the include file. This had the advantage of automatically generating and numbering the template parameters, reducing errors caused by manually editing the macros. The code is still easy to read in the python script, as it uses the handy python feature of named string interpolation from a dictionary:

```
print "The value is %(my_integer)d." % my_dictionary
```

The template parameters are constructed and put into a dictionary, which is then used to "fill out" the templated code in a multi-line string. This is similar to how the macro processor did the job.

The python script uses a user-written configuration file to define the fundamental units, derived units, and constants that the user feels are important to the problem at hand. This lets the library adapt to the user's problem domain, not the other way around. This is important, as it lets the user think about the problem in terms familiar to him.

An advantage of the configuration file is that changes to it can be made at any time and a new include file implementing the library can be quickly generated. No important existing work is lost or needs to be redone. For example, you can easily add a new fundamental unit. Behind the scenes, all the templated types change, but everything else you've already defined continues to work. This is maintenance Nirvana -- or at least it's not maintenance hell.

The configuration file lets you choose the numerical type you want to use for your calculations. It defaults to `double`, but you can use integers, floating point types, or any object type that has the proper numerical semantics. It should conceivably work with e.g. a multiple-precision numerical type that supports interval arithmetic, leading to sophisticated numerical calculations that still have simple semantics.

The python script `unitscpp.py` uses the `-e` option to send an empty configuration file to stdout. The resulting file has numerous comments to show how to construct a configuration file.

On the correct path

While implementing this library, I came to a point in writing examples and tests where the library was finding my mistakes much more frequently than I was finding implementation mistakes in the library. From my past experience in writing software, this is a strong indication that I was on track for developing a useful tool. I suspect if you give the library a try, you'll have similar experiences.

Another indicator of a useful tool is that I found it quite pleasurable to write and read code that used the library. It was pleasing to find that code that looked correct was correct. With care, it makes it much easier to write numerical code that works the first time.

Stroustrup's description of the development process (section 23.4 of [Strous1997]) describes exactly how this project evolved. The desired characteristics I gave in the table above didn't magically appear all at once -- they evolved iteratively from earlier versions and introspection.

Overview of the library

The `unitscpp.py` script is used to generate a problem-specific include file that provides the desired unit types to a particular program. The contents of the include file are simple, once you filter out the "textual noise" of numerous template parameter lists. The class used for the units contains the following member functions (`NT` refers to the number type used with the class, typically `double`):

`operator()`	Returns the `NT` value of the number represented by the class. Thus, if you have a Unit object `xyz`, you can get the "raw" numerical value of that object by e.g. `double my_value = xyz();`
`dim`	Returns a string representing the dimensions of the unit. For example, if the units class consisted of the three fundamental units mass, length, and time, a velocity's dimensions would be returned as `<0,1,-1>` which represents $M^0 L^1 T^{-1}$.
`to`	Convert to a specified (and compatible) unit and return the value as an `NT` value. Typical use might be (where `x` is in meters as the fundamental unit): `cout << "x is " << x.to(mm) << " mm." << endl;` Of course, the unit `mm` will have to be defined.
`operator-=, +=`	Addition and subtraction with other unit objects.
`operator*=, /=`	Multiplication and division with `NT` types.

Functions are provided to allow the following operations

`operator+, -, *, /`	Multiplication and division with unit and NT types.
`operator<, >, <=, >=`	Ordering.
`operator==`	Equality testing.
`operator<<`	Insertion operator for streams.

That's the extent of the functionality. Most functions are one line of code; it's a lightweight library.

Library Usage

Here are the steps to use the library:

1. Write a configuration file to define fundamental units, derived units, and constants pertinent to your problem's domain.
2. Generate the include file.
3. Write your code using the units you've defined.

Configuration files

The `unitscpp.py` script reads the configuration file and generates the include file containing the code that defines the unit types. You can use the `-e` option to `unitscpp.py` to get an empty configuration file with comments explaining how to construct the file. Use the `-s` option to get a configuration file with a number of SI units defined (you can customize this output to your tastes by editing the `unitscpp.py` file).

Lines beginning with `#` (whitespace before the `#` symbol is allowed) are comments and are ignored. Since the comments gotten with the `-e` option explain the details, I'll just cover the key pieces of information needed.

Each line begins with a token (tokens are separated by whitespace) that defines the line type. There are a few that describe details like defining the (optional) namespace to put things in, the name of the templated class implementing the units types, the name of the include file, and the specification of the numerical type to use.

The main tokens used are:

`Unit`

Defines a fundamental unit. All derived units will ultimately be expressed in terms of these fundamental units. Some typical lines might be

```
Unit = Mass
Unit = Length
Unit = Time
```

`DerivedUnit`

Defines a unit that is derived from the fundamental units. You may write expressions with `*`, `/`, `^`, `**`, and parentheses for grouping. `^` and `**` mean exponentiation with an integer exponent. Some example lines are

```
DerivedUnit = Velocity = Length/Time
DerivedUnit = Acceleration = Velocity/Time
DerivedUnit = Force = Mass*Acceleration
DerivedUnit = Pressure = Force/Length^2
DerivedUnit = Stress = Pressure
DerivedUnit = Energy = Force*Length
DerivedUnit = Frequency = 1/Time
```

Such expressions allow you to define units in a natural way.

`Constant`

These lines are intended to be used with `const` expressions defining unit values. However, the code after the `=` symbol is copied verbatim to the include file, so any legal C++ code is allowed. Some example lines are

```
Constant = const Length m = 1;
Constant = const Time s = 1;
Constant = const Time hour = 3600*s;
Constant = const Length mm = 1e-3*m;
Constant = const Length um = 1e-6*m;
Constant = const Length nm = 1e-9*m;
Constant = const Length inch = 25.4*mm;
Constant = const Velocity c = 3e8*m/s;
```

You only need to define the constants you want to use in your program. Note that physical constants are defined the same way as "regular" units. This lets an expression like `v.to(c)` convert a velocity to a fraction of the speed of light in an intuitive way. You can also write conversions such as `v.to(nm/hour)` and it's obvious what's going on. Always remember, however, that the `to` method returns an `NT` type, not a units type.

There are also constructs to allow you to include arbitrary code at the beginning and the end of the include file.

Example: Barlow's Formula

This example shows the code for a simple numerical calculation using Barlow's formula for thin-walled cylinders. It relates the stress, diameter, wall thickness, and fluid pressure inside a pipe. The formula is

$$t = \frac{p\,D}{2(S+p)}$$

where

t = wall thickness in inches
p = fluid pressure in pipe in psi
D = pipe outside diameter
S = allowable stress in psi

I chose this example in non-SI units to show that you can do the calculations with SI units or define the fundamental units as length and pressure.

Let's solve for the case of a 1 inch diameter 304 stainless steel pipe with a yield strength of 30 kpsi. We want the allowed stress to be one half of this yield strength, so S = 15000 psi. The working pressure will be 500 psi. Plugging the numbers in gives t = 500/31000 = 0.016 inches.

Calculation using SI units

We'd write the following configuration file (some leading and trailing lines are not shown):

```
Unit = Length
Unit = Mass
Unit = Time

DerivedUnit = Area = Length*Length
DerivedUnit = Acceleration = Length/Time**2
DerivedUnit = Force = Mass*Acceleration
DerivedUnit = Pressure = Force/Area
DerivedUnit = Stress = Pressure

Constant = const Length m = 1;
Constant = const Time s = 1;

Constant = const Length mm = 1e-3*m;
Constant = const Length inch = 25.4*mm;
Constant = const Force N = 1;
Constant = const Force lbf = 4.4482216*N;
Constant = const Pressure psi = lbf/(inch*inch);

Constant = typedef Length WallThickness;
Constant = typedef Length Diameter;
```

Note the `typedef`s of WallThickness and Diameter. These aren't strictly needed, but they help to make our code self-documenting.

We'd run the `unitscpp.py` script with this configuration file. Let's suppose we saved the output to a file named `units.h`.

Then we'd put the following code into the file `barlow.cpp`:

```
#include <iostream>
#include <iomanip>
using namespace std;

#include "units.h"
```

```
WallThickness Barlow
(
    const Pressure & p,
    const Diameter & OD,
    const Stress & S
)
{
    return p*OD/(2*(S + p));
}

int main(void)
{
    const Pressure p = 500*psi;
    const Diameter OD = 1*inch;
    const Stress S = 15e3*psi;
    const WallThickness t = Barlow(p, OD, S);
    cout << "304 SST pipe wall thickness = " << t.to(inch)
         << " inches" << endl;
}
```

I feel this is quite readable code with the advantage of having the compiler check to make sure you don't make errors in calculations due to variables or constants having the wrong units. Normally, I demand descriptive variable names, but the short names make it easy to see the structure of Barlow's equation (and compare it to the printed form). The variable types add significant information for the reader.

An advantage of defining units above in SI is that the metric equivalents are easily gotten if desired.

Calculation using direct units

Some engineers might prefer to see the calculations done directly in inches and pressure. This is easily done with the following configuration file:

```
Unit = Length
Unit = Pressure

Constant = const Length inch = 1;
Constant = const Pressure psi = 1;

Constant = typedef Pressure Stress;
Constant = typedef Length WallThickness;
Constant = typedef Length Diameter;
```

These simple definitions will yield the identical answer as the previous example (note `Stress` was changed from a derived unit to a `typedef`; the results are still the same). This demonstrates a strength of the UnitsC++ library: only define what you need and fit the library to the problem, not the other way around.

Summary

The UnitsC++ units library has been presented that should be easy to use, modify, and understand. Its main advantages are:

1. It makes code easier to read and understand.
2. It helps make C++ numerical code easier to make correct by using the compiler to catch dimension errors in calculations.
3. It's lightweight. Once you're familiar with it, all you need to have on hand is the 30 kilobyte `unitscpp.py` script. There's essentially no runtime penalty in space or time for using the library. The size of a units object is the same as the numerical type you've chosen to use

with it. (I said essentially because some compilers might use extra space for the constant definitions in multiple compilation units.)

References

[BN1994]	J. Barton and L. Nackman, *Scientific and Engineering C++*, Addison-Wesley, 1994, ISBN 0201533936. These folks had the insight of using templates with integer parameters for dimensional calculations in C++ programs (see section 16.5).
[Brown2001]	W. Brown, "Applied Template Metaprogramming in SIunits: the Library of Unit-Based Computation", 2001. http://www.oonumerics.org/tmpw01/brown.pdf. This elegant paper mentioned that the author was going to submit the library to Boost, then make a proposal for its inclusion into the C++ standard. Sadly, these things apparently have not happened. I have not been able to get a copy of the code to SIunits.
[GNUunits]	GNU units program, http://www.gnu.org/software/units/,
[Ham2004]	B. Hamilton, "Measurement Calculus" https://jmci.dev.java.net/files/documents/1529/5908/MeasCalculusWhitePaper.pdf, June 28, 2004. Discusses why dealing with dimensions in a programming language can be harder than it looks.
[Java2001]	http://www.jcp.org/en/jsr/detail?id=108. Java Specification Request for support of units; withdrawn apparently due to apathy.
[Kenn2002]	M. Kenniston, "Dimension Checking of Physical Quantities", http://www.ddj.com/architect/184401579, November 1, 2002. A 2001 implementation exists, but is dormant and the author has indicated that he's too busy to pursue further development.
[Meyers2006]	S. Meyers, "My Most Important C++ Aha! Moments...Ever", http://www.artima.com/cppsource/top_cpp_aha_moments.html. Acknowledges the insight of Barton and Nackman's. Meyers' stuff is always worth reading.
[NASA1999]	http://www.cnn.com/TECH/space/9909/30/mars.metric/
[Pad2007]	A. Padmanabh http://learningcppisfun.blogspot.com/2007/01/units-and-dimensions-with-c-templates.html, January 24, 2007.
[SFmcs::units]	Sourceforge project mcs::units: http://sourceforge.net/projects/mcs-units/
[SFquantities]	B. Speiser, "Quantities - A Collection of C++ Classes to Handle Quantity Calculus in C++", https://sourceforge.net/projects/quantity/, 2007.
[SFtuoml]	Sourceforge project The Units of Measure Library: http://sourceforge.net/projects/tuoml/
[Strous1997]	B. Stroustrup, *The C++ Programming Language*, 3rd. ed., Addison-Wesley, 1997. By the master and a source of inexhaustible insight.
[Unum2004]	http://home.scarlet.be/be052320/Unum_diary.html. A python facility for doing calculations with numbers with units.