

# Re: Dealing with ad hominem attacks in comp.programming

---

*Source:* <http://coding.derkeiler.com/Archive/General/comp.programming/2008-02/msg01141.html>

---

- *From:* kwikius <[andy@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx](mailto:andy@xxxxxxxxxxxxxxxxxxxxxxxxxxxxx)>
  - *Date:* Wed, 20 Feb 2008 06:50:07 -0800 (PST)
- 

On Feb 20, 6:56 am, "Clive D. W. Feather" <[cl...@on-the-train.demon.co.uk](mailto:cl...@on-the-train.demon.co.uk)> wrote:

<...>

An alternative would be to represent quantities as a floating point number and an array of unit exponents. That is, working with the SI units of m, s, kg, K, A, cd, you'd represent:

3 m as { 3.0, 1, 0, 0, 0, 0, 0 }

3 s as { 3.0, 0, 1, 0, 0, 0, 0 }

3 m/s as { 3.0, 1, -1, 0, 0, 0, 0 }

and so on. The + and - operators would check that the units are the same and throw an exception if not, while the \* and / operators would manipulate the units accordingly.

The big advantage of this is that it's open-ended: if someone needs to work with  $\text{kg}^2/\text{Acd}^3$ , you don't need another set of overloaded operators to do it. The big disadvantage, of course, is that you lose compile-time checking of consistency.

Following is a very cut down version of how the dimensional analysis part of quan works. This is hugely simplified and to do anything more you would have to add suitable stuff for subtraction etc of compile time rationals and dimensions, but there should be enough here to get the basic idea. Note that only one overloaded operator \* is reqd. It will work for any dimension combo.

Also note that C++ templates can be used to do arbitrarily complex computations, as in gcd "metfunction" here.

Also shows some of the problems with metaprogramming in C++. its efectively assembly language...

regards  
Andy Little

## Re: Dealing with ad hominem attacks in comp.programming

```
// very cut down and simple quantity type

// in meta namespace nothing is instantiated
// its all type building stuff
namespace meta{
/*
compile time gcd uses compile time recursion
*/
template <int N, int D>
struct gcd : gcd<D,(N%D)>::type{ };

template <int N>
struct gcd<N,0>{
typedef gcd type;
static const int value = N;
};
/*
compile time rational number
N.B dont think it handles negatives here
*/
template <int N, int D>
struct rational{
static const int gcd_ = gcd<N,D>::value;
static const int nume = N / gcd_;
static const int denom = D / gcd_;
typedef rational<nume,denom> type;
};

/*
compile time addition "metafunction"
*/
template <typename Lhs, typename Rhs>
struct plus;

/**overload* plus for rational
template <int NL,int DL, int NR, int DR>
struct plus<
rational<NL,DL>,rational<NR,DR>
>{
//n.b normalised result
typedef typename rational<
(NL * DR + NR * DL),(DL * DR)
>::type type;
};

// for simplicity only 3 dims used here
// all params are rationals
template <typename T1, typename T2, typename T3>
struct dimension{
typedef T1 pow_length;
typedef T2 pow_time;
```

## Re: Dealing with ad hominem attacks in comp.programming

```
typedef T3 pow_mass;
};

// overload add for dimension
template <
typename T1L, typename T2L, typename T3L,
typename T1R, typename T2R, typename T3R
>
struct plus<
dimension<T1L,T2L,T3L>,
dimension<T1R,T2R,T3R>
>{
typedef dimension<
typename plus<T1L,T1R>::type,
typename plus<T2L,T2R>::type,
typename plus<T3L,T3R>::type
> type;
};

} //meta

// very basic physical quantity
// D param is a meta::dimension<..>

template <typename D>
struct quantity{
explicit quantity(double const & in) : v(in){};
double v;
};

template <typename DL, typename DR>
inline
quantity<typename meta::plus<DL,DR>::type>
operator *(
quantity<DL> const & lhs,
quantity<DR> const & rhs
)
{
return quantity<
typename meta::plus<DL,DR>::type
>(lhs.v * rhs.v);
}

#include <iostream>
int main()
{
typedef meta::rational<0,1> zero;
typedef meta::rational<1,1> one;

typedef meta::dimension<one,zero,zero> dlength;
typedef meta::plus<dlength,dlength>::type darea;
```

## Re: Dealing with ad hominem attacks in comp.programming

```
typedef meta::plus<dlength,darea>::type dvolume;
```

```
typedef quantity<dlength> length;  
typedef quantity<darea> area;  
typedef quantity<dvolume> volume;
```

```
length x(1);  
length y(2);
```

```
area a = x * y;  
volume v = a * y;
```

```
std::cout << sizeof(v) << "\n";  
std::cout << sizeof(double) << "\n";
```

```
#if(0)  
//*****  
v = a * a; // Error  
length xx = x * y; // Error  
//*****  
#endif  
  
}
```

.