Dimensional Analysis in C++

Scott Meyers, Ph.D.
Software Development Consultant

smeyers@aristeia.com
http://www.aristeia.com/

Voice: 503/638-6028
Fax: 503/638-6614

Dimensional Analysis in C++

Scientific and engineering calculations are dependent on correct use of units in calculations:

- It makes no sense to assign a time value to a distance variable
- It makes no sense to compare a mass variable with a charge variable

But most software ignores such units:

```
double t;                // time - in seconds
double a;                // acceleration - in meters/second²
double d;                // distance - in meters
...
cout << d/(t*t) - a;     // okay, subtracts meters/sec²
cout << d/t - a;         // should be an error, as it
                         // subtracts meters/sec and
                         // meters/sec²
```

Dimensional Analysis in C++

Typedefs just disguise the problem:

```
typedef double Acceleration;
typedef double Time;
typedef double Distance;

Time t;
Acceleration a;
Distance d;

...

cout << d/t - a;                    // still compiles, but is still wrong
```

We want a way to use the C++ type system to:

- Make unit compatibility errors impossible:
  - They'll be detected during compilation
- Do so with minimal runtime performance impact:
  - Minimal memory overhead, minimal runtime overhead
  - As much as possible should be done during compilation

Enforcing Dimensional Unit Correctness

Observations:

- The number of needed types is, in principle, unlimited:
  - Time * Time = Time^2
  - Time/Distance = Time/Distance
  - Distance/Time^2 = Distance/Time^2
- This suggests we should have templates generate the types automatically.
- Types change only when a unit type's *exponent* changes:
  - Unitless numbers (i.e. constants) have unit exponents of 0
  - In Time * Time, the Time exponent goes from 1 to 2
  - In Acceleration/Time, the Time exponent goes from -2 to -3
- This suggests we need a template to generate types based on unit exponents

Enforcing Dimensional Unit Correctness

```
template<int m,                    // exponent for mass
         int d,                    // exponent for distance
         int t>                    // exponent for time
class Units {
public:
  explicit Units(double initVal = 0): val(initVal) {}

  double value() const { return val; }
  double& value() { return val; }
  ...

private:
  double val;
};
```

Now we can say:

```
Units<1, 0, 0> m;                  // m is of type mass
Units<0, 1, 0> d;                  // d is of type distance
Units<0, 0, 1> t;                  // t is of type time

m = t;                             // error! type mismatch
```


Enforcing Dimensional Unit Correctness

Typedefs for commonly-used units make things clearer:

```
typedef Units<1, 0, 0> Mass;
typedef Units<0, 1, 0> Distance;
typedef Units<0, 0, 1> Time;

Mass m;
Distance d;
Time t;
```

Enforcing Dimensional Unit Correctness

Arithmetic operations on these kinds of types are important, so we can augment Units as follows:

```
template<int m, int d, int t>
class Units {
public:
  ...                                        // as before

  Units<m, d, t>& operator+=(const Units<m, d, t>& rhs)
  {
    val += rhs.val;
    return *this;
  }

  Units<m, d, t>& operator*=(double rhs)
  {
    val *= rhs;
    return *this;
  }

  ...
};
```

Operators for subtraction and division are analogous.

Enforcing Dimensional Unit Correctness

Non-assignment operators are best implemented as non-members:

```
template<int m, int d, int t>
const Units<m, d, t> operator+(const Units<m, d, t>& lhs,
                               const Units<m, d, t>& rhs)
{
  Units<m, d, t> result(lhs);
  return result += rhs;
}

template<int m, int d, int t>
const Units<m, d, t> operator*(double lhs,
                               const Units<m, d, t>& rhs)
{
  Units<m, d, t> result(rhs);
  return result *= lhs;
}

template<int m, int d, int t>
const Units<m, d, t> operator*(const Units<m, d, t>& lhs,
                               double rhs)
{
  Units<m, d, t> result(lhs);
  return result *= rhs;
}
```

Enforcing Dimensional Unit Correctness

If we adopt the SI units as our standard, we can provide the following constants:

```
const Mass kilogram(1);        // each of these constants sets its
const Distance meter(1);       // internal val field to 1.0
const Time second(1);
```

Now we can start defining more interesting objects:

```
Distance myBatikHeight(0.5 * meter);
Distance myBatikWidth(1 * meter);

Mass myWeight(88.6 * kilogram);

Time halfAMinute(30 * second);
```


Enforcing Dimensional Unit Correctness

We can also define other units in terms of our standard:

```
const Mass pound(kilogram/2.2);

const Mass ton(907.18 * kilogram);

const Time minute(60 * second);

const Time hour(60 * minute);

const Time day(24 * hour);

const Distance inch(.0254 * meter);
```

Enforcing Dimensional Unit Correctness

The real fun comes when multiplying/dividing Units:

```
template< int m1, int d1, int t1,
          int m2, int d2, int t2>
const Units<m1+m2, d1+d2, t1+t2>
operator*( const Units<m1, d1, t1>& lhs,
           const Units<m2, d2, t2>& rhs)
{
  typedef Units<m1+m2, d1+d2, t1+t2> ResultType;

  return ResultType(lhs.value() * rhs.value());
}

template< int m1, int d1, int t1,
          int m2, int d2, int t2>
const Units<m1-m2, d1-d2, t1-t2>
operator/( const Units<m1, d1, t1>& lhs,
           const Units<m2, d2, t2>& rhs)
{
  typedef Units<m1-m2, d1-d2, t1-t2> ResultType;

  return ResultType(lhs.value() / rhs.value());
}
```


Enforcing Dimensional Unit Correctness

Real implementations typically use more template arguments for Units:

- One specifies the precision of the value (typically float or double)
- The others are for the exponents of the seven SI units:
  - Mass
  - Length
  - Time
  - Charge
  - Temperature
  - Intensity
  - Angle

Enforcing Dimensional Unit Correctness

```
template<class T, int m, int d, int t, int q, int k, int i, int a>
class Units {
public:
  explicit Units(T initVal = 0) : val(initVal) {}
  T& value() { return val; }
  const T& value() const { return val; }
  ...
private:
  T val;
};

template<class T, int m1, int d1, int t1, int q1, int k1, int i1, int a1,
                  int m2, int d2, int t2, int q2, int k2, int i2, int a2>
Units<T, m1+m2, d1+d2, t1+t2, q1+q2, k1+k2, i1+i2, a1+a2>
operator*(const Units<T, m1, d1, t1, q1, k1, i1, a1>& lhs,
          const Units<T, m2, d2, t2, q2, k2, i2, a2>& rhs)
{
  typedef Units<T, m1+m2, d1+d2, t1+t2, q1+q2, k1+k2, i1+i2, a1+a2>
          ResultType;

  return ResultType(lhs.value() * rhs.value());
}
```


Observations

Dimensionless quantities (i.e., objects of type Units<T, 0,0,0,0,0,0,0>) should be type-compatible with unitless types (e.g., int, double, etc.).

- Partial template specialization can help:

```
template<typename T>
class Units<T, 0, 0, 0, 0, 0, 0, 0> {
public:
  ...
  Units(T initVal = 0): val(initVal) {}      // allow implicit conversion
  operator T() const { return val; }         // to/from values of type T

  Units& operator=(T newVal)                 // allow assignments from
  { val = newVal; return *this; }            // values of type T
  ...

private:
  T val;
};
```

If partial template specialization is unavailable, you can totally specialize for e.g., T = double and/or T = float.

Observations

Some compilers refuse to place objects in registers:

- A Units<double, ...> may thus be treated less efficiently than a raw double
- If efficiency is a problem, you can revert to type-unsafe typedefs:

```
typedef double Acceleration;
typedef double Time;
typedef double Distance;
```

  - ➠ This is okay as long as the code has already been shown to compile using Units

Observations

A state-of-the-art implementation of the Units approach is more efficient, powerful, and sophisticated:

- It allows fractional exponents (e.g., distance$^{1/2}$).
- It supports multiple unit system views (beyond basic SI).
- It puts all exponent parameters into a struct to improve the readability of the code.

Conclusions

- Templates are useful for a lot more than just containers
- Templates make it possible to generate and check an unknowable number of types during compilation
- Templates can add type safety to code with little or no runtime penalty

Further Reading

- John J. Barton and Lee R. Nackman, "Dimensional analysis," *C++ Report*, January 1995. Based on section 16.5 of their *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, Addison-Wesley, 1994, ISBN 0-201-53393-6.
  - ➠ Now primarily of historical interest.
- Walter E. Brown, "Introduction to the SI Library of Unit-Based Computation," International Conference on Computing in High Energy Physics (CHEP '98), August 1998. Available at http://fnalpubs.fnal.gov/archive/1998/conf/Conf-98-328.pdf.
  - ➠ A user's view of SIUNITS. Describes how five different models of the universe are supported.
- Walter E. Brown, "Applied Template Metaprogramming in SIUNITS: the Library of Unit-Based Computation," Second Workshop on C++ Template Programming, October 2001. Available at http://www.oonumerics.org/tmpw01/brown.pdf.
  - ➠ Another description of SIUNITS, this time focusing more on implementation strategies.

Further Reading

- Michael Kenniston, "Dimension Checking of Physical Quantities," *C/C++ Users Journal*, November 2002.
  - A description of a slightly different approach, one focused on working with less conformant compilers (e.g., Visual C++ 6).