Scientific and Engineering C++: Dimensional Analysis

John J. Barton Lee R. Nackman

October 31, 1994

Scientists and engineers routinely apply *dimensional analysis* to check formulas for consistent units. Each quantity in a formula is assigned a unit and, for dimensonally correct forms, arithmetic on the units yields the same result on both sides of the equation. For example, we can check the formula $d = \frac{1}{2}at^2$ for the distance d traveled by a body under constant acceleration a in a time t: the units of acceleration (a) are meters per second squared, the unit of time (t) is the second, so the units of at^2 are

$$\frac{\text{meters}}{\text{second}^2} \times \text{second}^2.$$

The seconds "cancel out" and $\frac{1}{2}$ is dimensionless, so both sides of the equation have units of meters. Everything's cool. But try the same thing with the equation $d = at$ and the units don't match; the equation must be wrong.

Most of us do this kind of thinking routinely whenever we work with equations involving physical quantities. Yet, once we transcribe our thinking into code, we throw out the dimensions and write

```
double a;     // Acceleration a,
double t;     // for time t, gives
double d;     // distance d
// ...
d = .5 * a * t * t;
```

At most we might use typedefs to document the units, like this:

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

Acceleration a;
Time         t;
Distance     d;
// ...
d = .5 * a * t * t;
```

Unfortunately, this is only documentation; the code

```
d = .5 * a * t;
```

runs just fine, but does a nonsensical assignment.

Dimensional analysis is a form of type-checking: the product of an acceleration and a time is a velocity, not a distance. When we compute a physical formula with the double builtin datatype, we are computing without types. In this column, we combine class templates and function templates to provide dimensional analysis at compile time by creating new datatypes for physical quantities with units. Make a mistake with units and the C++ type checking will give you a compile error. As we'll see, this can be done with no space overhead and little or no execution-time overhead. Along the way, we'll see some unconventional but powerful uses of templates. Similar approaches to this problem have been discussed by Umrigar [1] and in Section 16.5 of our book [2].

Measurement Systems. A measurement with a ruler might yield a result like 1 centimeter. This result has three parts. There is a *value*: 1. There is a *dimension*: we are measuring length, not mass. And there is a *unit*: we measure length in terms of centimeters not inches or miles. Without dimensonal analysis in our code we would represent a measurement with just a floating point number for the value. To do better we need to understand dimensions and units.

A measurement system is based on a set of *fundamental dimensions*. There are many choices [3, 4]. The most common measurement systems today are based on dimensions of mass, length, time, charge, temperature, intensity and angle. For simplicity, we'll work with only the fundamental dimensions of mass, length, and time and denote them by M, L, and T, respectively.

Given the fundamental dimensions, one forms *derived dimensions* as products of powers of the fundamental dimensions based on definitions or physical laws. For example, velocity is defined as the derivative of distance with respect to time. Distance is an increment of length, so velocity has dimensions $[L/T]$. Acceleration is defined to be the derivative of velocity with respect to time, so it has dimensions $[(L/T)/T]$ or $[L/T^2]$. Newton's law tells us that $F = ma$, so force has dimensions of $[M] \times [L/T^2]$ or $[ML/T^2]$. Table 1 gives the dimensions of some common physical quantities. Notice that Poisson's ratio is a dimensionless quantity: the exponents of all its dimensions are zero.

For each dimension, we must select a standard quantity or *unit* corresponding to a 1 value in that dimension. We measure length in inches, or centimeters, or miles, or furlongs: that means we compare lengths to standard units of length called an inch, a centimeter, a mile or a furlong, respectively. Likewise, we must choose units

Quantity	Dimensions
Velocity	$[LT^{-1}]$
Acceleration	$[LT^{-2}]$
Force	$[MLT^{-2}]$
Pressure	$[ML^{-1}T^{-2}]$
Energy	$[ML^2T-2]$
Momentum	$[MLT^{-1}]$
Power	$[ML^2T^{-3}]$
Poisson's ratio	$[1]$

Table 1: Dimensions of common physical quantities.

for derived dimensions: furlongs per fortnight, miles per hour, or meters per second.

We'll adopt the dimensions and units defined by the international standard called *Système International*, abbreviated *SI*. In the SI system, the unit of length is the meter, the unit of mass is the kilogram, and the unit of time is the second.

Physical Quantities. Let's call the combination of a value and a unit a *physical quantity* and let's write a class to represent them. Since we want to use this class to represent any physical quantity that appears in our programs, it needs to be both space and time efficient. We'd like to be able to have arrays of physical quantities, so our representation should not impose space overhead. Thus we can only store a quantity's value; its unit or dimension information will be encoded in its data type.

These data types exist to check for errors like adding a mass and a velocity. The dimensional analysis "rules" are simple:

1. Two physical quantities can be added, subtracted, or compared if and only if they have the same dimensions and are expressed in terms of the same units.

2. The product (quotient) of any two physical quantities is computed by multiplying (dividing) the dimensions, units, and value.

We want the C++ type system to apply these rules at compile time to expressions we construct from physical quantities.

We'll need one type for each possible set of dimensions, a potentially infinite set of types. This set of types can be expressed by a class template with integer template arguments for the dimension exponents:

```
template<
  int m,      // Mass exponent
  int l,      // Length exponent
  int t       // Time exponent
  >
class Physical {
public:
  Physical() : val(1.0) {}  // A unit
  double  value() const { return val; }
  double& value()       { return val; }
private:
  typedef Physical<m, l, t> Phys; // shorthand
public:
  Phys& operator+=(Phys rhs) {
    val += rhs.val; return *this;
  }
  Phys& operator-=(Phys rhs) {
    val -= rhs.val; return *this;
  }
  Phys& operator*=(double rhs) {
    val *= rhs; return *this;
  }
  Phys& operator/=(double rhs) {
    val /= rhs; return *this;
  }
  Phys convertTo(Phys dest) const;
  static Phys unit;  // Shorthand.
private:
  double val;
};
```

No constructor taking a single `double` is provided: it would be a conversion operator allowing dimensionally incorrect assignment of a dimensionless quantity (the `double`) to any physical.

The `+=` and `-=` assignment operators implement addition and subtraction of physical quantities with the same dimensions, and the `*=` and `/=` assignment operators implement multiplication and division by dimensionless quantities. There are no assignment operators for self-type products (and quotients) because this product's dimensions would be different than the dimensions of its operands. For example, `a*=b` cannot work if `a` and `b` both have units of `inch`. The `convertTo()` member function, which converts a quantity from one unit to another, is described later.

Global functions implement the corresponding non-assignment operators. Here are a few of them:

```
template<int m, int l, int t>
Physical<m, l, t>
operator+(Physical<m, l, t> lhs,
          Physical<m, l, t> rhs) {
  Physical<m, l, t> result = lhs;
  return result += rhs;
}

template<int m, int l, int t>
Physical<m, l, t>
operator*(double lhs, Physical<m, l, t> rhs) {
  Physical<m, l, t> result = rhs;
  result.value() *= lhs;
  return result;
}

template<int m, int l, int t>
Physical<m, l, t>
operator*(Physical<m, l, t> lhs, double rhs) {
  Physical<m, l, t> result = lhs;
  result.value() *= rhs;
  return result;
}
```

```
template<int m, int l, int t>
Physical<m, l, t> Physical<m, l, t>::unit;
```

Notice that the operators that take a dimensionless quantity are asymmetric and two versions are needed, one with the dimensionless quantity as the left operand, the other with it as the right operand.

Products and quotients of two physical quantities are provided by function templates that "compute" the return type (at compile time):

```
template<
  int m1, int l1, int t1,
  int m2, int l2, int t2
  >
Physical<m1+m2, l1+l2, t1+t2>
operator*(Physical<m1, l1, t1> lhs,
          Physical<m2, l2, t2> rhs) {
  return Physical<m1+m2, l1+l2, t1+t2>::unit *
        lhs.value() * rhs.value();
}

template<
  int m1, int l1, int t1,
  int m2, int l2, int t2
  >
Physical<m1-m2, l1-l2, t1-t2>
operator/(Physical<m1, l1, t1> lhs,
          Physical<m2, l2, t2> rhs) {
  return Physical<m1-m2, l1-l2, t1-t2>::unit *
     lhs.value() / rhs.value();
}
```

The `return` expression begins with the `static` object `unit` with the correct dimensions and scales it with the value of the new physical quantity.

Finally, let's provide an output operator:

```
template<int m, int l, int t>
ostream&
operator<<(ostream& os, Physical<m, l, t> q) {
  os << q.value();
  if (m != 0 || l != 0 || t != 0) os << '[';
  if (m != 0) os << "M^" << m;
  if (l != 0) os << "L^" << l;
  if (t != 0) os << "T^" << t;
  if (m != 0 || l != 0 || t != 0) os << ']';
  return os;
}
```

As an example, here is a function that computes the distance that a dropped object falls, ignoring friction:

```
Physical<0, 1, 0>
dist_fallen(Physical<0, 0, 1> t) {
  // Distance a dropped object falls in time t,
  // ignoring friction.
  const Physical<0, 1, -2> g =
          9.80665 * Physical<0, 1, -2>::unit;
  return .5 * g * t * t;
}
```

This function implements the equation $d = \frac{1}{2}at^2$. The dimensions are checked at compile time, but the code is ugly and hard to read.

Some Prettier Names. A few `typedef`s for the fundamental dimensions and some derived dimensions helps code readability:

```
class FundamentalDimensions {
public:
  typedef Physical<1, 0, 0> Mass;
  typedef Physical<0, 1, 0> Length;
  typedef Physical<0, 0, 1> Time;
};

class DerivedDimensions :
  public FundamentalDimensions {
public:
  typedef Physical<0, 1, -1> Velocity;
  typedef Physical<0, 1, -2> Acceleration;
  typedef Physical<1, 1, -2> Force;
  typedef Physical<1, 2, -2> Energy;
  // ...
};
```

Nesting the typedefs avoid name space conflicts. For example, the dimensional analysis types `Time` and `Energy` might conflict with other type names if they were at global scope.

Physical constants often appear in formulas, so we add one more class that defines common constants:

```
class PhysicalConstants :
  public DerivedDimensions {
public:
  static const Mass kilogram;
  static const Length meter;
  static const Length centimeter;
  static const Length inch;
  static const Time second;
  static const Time minute;
  static const Time hour;
  static const Acceleration meter_per_sec2;
  static const Acceleration g;
  // ...
};
```

Definitions for the units of the measurement system, like `kilogram`, `meter`, and `second`, allow obvious code to be used for dimension specifications:

```
const PhysicalConstants::Acceleration
      PhysicalConstants::g
              = 9.80665*meter/(second*second);
```

These `typedef`s and constants can be used by any member of a class derived from the appropriate class:

```
class FallingObject :
  public PhysicalConstants {
public:
  // ...
  Length distance(Time t) {
    return .5 * g * t * t;
  }
};
```

The `typedef`s are inherited by `FallingObject`, so its member functions can use the names defined by the typedefs. Similarly the constants are in scope and can be used simply by name. The resulting expression is as simple as the one we have written without dimensional analysis, but this one is checked for us.

Converting Units. Earlier we decided to represent all physical quantities in terms of the SI system of units. In practice, people use inches, centimeters, etc., not just meters. The convertTo() member does the necessary conversion:

```
template<int m, int l, int t>
Physical<m, l, t>
Physical<m, l, t>::
convertTo(Physical<m, l, t> dest) const {
  return unit * (val / dest.val);
}
```

Here is an example of how it can be used to convert meters to inches:

```
  obj.distance(2.0*second).convertTo(inch)
```

Other operations. The templates we used here generate types through addition of constant expressions. Division by constants can be used to support operations like square root:

```
template<int m, int l, int t>
Physical<m/2, l/2, t/2>
sqrt(Physical<m, l, t> x) {
  assert(m%2 == 0);
  assert(l%2 == 0);
  assert(t%2 == 0);
  return Physical<m/2,l/2,t/2>::unit *
                        sqrt(x.value());
}
```

We added the calls to assert() because the C++ rules for constant expressions apply division and truncation before determining the type of the expression. Thus no compile error is generated in this code even though the input type is Physical<0,1,0>:

```
cout << sqrt(1.0*inch) << endl; // fails at run-time
```

The type of the object returned by sqrt() would be Physical<0,0,0>; the calls to assert() insure that we get at least a run-time error. If the assert() tests prove to be any significant fraction of the numerical computation time, template specialization can be used for common values of the exponents:

```
Physical<0, 1, 0>
sqrt(Physical<0, 2, 0> x) {
 return sqrt(x.value())*Physical<0,1,0>::unit;
}
```

Our arithmetically generated types are limited to explicit equations. Implicit equations for the return type computation are not legal:

```
template<int m, int l, int t>
Physical<m, l, t>
sqrt(Physical<2*m, 2*l, 2*t> x) {
  // WRONG! Compile error.
  return sqrt(x.value());
}
```

Thus we cannot force the powers of the dimensions of the return type in the sqrt() template to be half that of the argument type by this means.

A Reality Check. Do these templates compile? Well, it depends on your compiler. Section 14.4c of the ARM [5] prohibits non-type template arguments for function templates, making all of our function templates illegal. However, changes in the language definition will eliminate this restriction (Section 15.6.1 of [6]). Some compilers enforce the ARM restriction, others don't. This code won't compile using a restrictive compiler.

Summary. Dimensional analysis was adopted for hand computation of physical formula for the same reason that type-checking was added to computer programming languages: to catch errors as early as possible. Computing with dimensional analysis is simply a matter of using the appropriate data types in computations and letting the C++ type system do the checking. Function templates returning types computed from the argument types make the construction of dimensional analysis data types especially easy.

References

[1] Zerksis D. Umrigar. Fully static dimensional analysis with C++. *ACM SIGPLAN Notices*, 29(9):135–139, September 1994.

[2] John J. Barton and Lee R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples.* Addison-Wesley, Reading, MA, 1994.

[3] Henry L. Langhaar. *Dimensional Analysis and Theory of Models.* John Wiley & Sons, New York, 1951.

[4] C. M. Focken. *Dimensional Methods and Their Applications.* Edward Arnold & Co., London, 1953.

[5] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, MA, 1990.

[6] Bjarne Stroustrup. *The Design and Evolution of C++.* Addison-Wesley, Reading, MA, 1994.