Applied Template Metaprogramming in SIUNITS: the Library of Unit-Based Computation

Walter E. Brown, Ph.D.
WB@fnal.gov
Computational Physics Department, Computing Division
Fermi National Accelerator Laboratory
Batavia, IL 60510-0500

June 15, 2001; revised August 23, 2001

Ye shall do no unrighteousness in judgment,
in measures of length, of weight, or of quantity.
— Leviticus 19:35

Abstract

While scientific programmers typically make heavy use of a programming language's native numeric data types, such practice has been a common source of errors: it obscures the diverse intentions (*e.g.*, distances, masses, energies, momenta, *etc.*) that any such purely numeric value could represent. Limitations in programming language expressiveness and compiler technology have historically made it difficult to address commensuration in a programming context. However, SIUNITS provides, in C++, a convenient means of computing with numeric values that have attached units.

SIUNITS applies compile-time type checking, thus avoiding run-time overhead. Via heavy use of template metaprogramming, SIUNITS provides data types corresponding to all base and derived dimensions specified by *le Système international d'Unités* (SI), the recognized international standard for describing measurable quantities and their units. Thus, for example, an object resulting from the product of two LENGTH objects is automatically of type AREA. As an important additional feature, SIUNITS permits arbitrary combinations of several provided measurement views, and even allows knowledgeable users to construct their own views.

Keywords: C++, SI, *Système international*, type checking, type safety, dimensional analysis, generic programming, template programming, metaprogramming.

Credit line: Work supported by the U.S. Department of Energy under contract No. DE-AC02-76CH03000.

Distribution: Approved for public release.

1 Dimensional analysis via type-checking

[A] famous bug in this shop is `2*pi+r*w` as the area of a strip of a cylinder; which escaped notice for years.
— Name withheld by request

Scientific work commonly deals with numbers that represent amounts of physical dimensions.[1] Scientists are therefore trained, as a matter of routine, to be very careful with these numbers' units, lest incommensurate (incompatible) values be accidentally combined in calculation or lest incorrect units be ascribed to the outcome. As Halliday and Resnick exhort [HR70, pp. 35-6]:

> In carrying out any calculation, always remember to attach the proper units.... One way to spot an erroneous equation is to check the dimensions of all its terms....

The methodology underlying this process is termed *dimensional analysis*. In modern digital computation, the analogous concept is known as *strong type checking*. This concept lies at the core of such object-oriented programming languages as C++, and yields among the most valuable benefits of such software methodology. To demonstrate the analogy's aptness, we paraphrase the above quotation (with differences **emphasized**):

> In **programming** any calculation, always remember to attach the proper **data type**. One way to spot an erroneous **program** is to check the **data types** of all its **objects**.

Given modern programming languages' significant expressive capabilities to design data types tailored to the problem domain at hand, we clearly now have both ample motivation and ample technology to embrace type-safe object-oriented techniques in all contemporary applications of computer programming. Alas, in computer programming as practiced today, the recommended standard of care seems only rarely applied to numeric quantities.

Informal inspection of contemporary production code samples revealed that, in numeric programming, programmers make heavy, near-exclusive, use of a language's native numeric types (*e.g.*, `double`). A 1998 search of representative on-line archives found not even a query on the subject, suggesting little or no interest in, or knowledge of, alternatives. In light of prior software art, this finding was not surprising.

As might be expected, such overly general practice is a common source of errors: it fails to distinguish the diverse intentions that any such purely numeric value could represent. For example, we can trivially form, in our programs, such capricious sums as Avogadro's number plus the speed of light. We can just as easily sum a DISTANCE with an ENERGY, or subtract a MASS from a MOMENTUM. Indeed, we could form, with no complaint from our computing environment, arbitrary meaningless combinations of incommensurate units.

Suppose, instead, that programmers routinely expressed, in their programs, the physical quantities and units that their software models. This practice would certainly yield "a succinct, intelligible form of program documentation, making programs easier to read and understand" while, more importantly, contemporary compiler technology would routinely check the computations for commensuration and provide "important feedback to the programmer by detecting a large class of program errors before execution" [Fag92, pp. ii, 2]. We hold that this process constitutes the computer-based equivalent of dimensional analysis: it achieves benefits consistent with those resulting from the unit-based approach so strongly advocated by Halliday and Resnick, among many others.

2 Introducing SIUNITS

2.1 Scope of the project

You will be able to appreciate the influence of such an Engine on the future progress of science.
— Charles Babbage

To address current unfortunate practices in numeric computation, we set out to develop a software subsystem to provide a convenient means of expressing, computing with, and displaying numeric values with attached units. We wished to obtain the well-known benefits of type safety consistent with recommended unit-based practices of long standing. An additional requirement of this project was to ensure strict compile-time type checking. More specifically, we sought:

[1] Some of this introductory material is adapted from [Bro98].

1. Application of current software technology to numeric physical concepts,
2. Convenient (near-trivial) expression in such application,
3. General utility rooted in existing standards,
4. Use of nomenclature from our problem domain, and
5. No attendant run-time performance (time or space) penalties!

Our project, SIUNITS: THE LIBRARY OF UNIT-BASED COMPUTATION, has succeeded in addressing these requirements. The resulting software module meets (and, in many respects, greatly exceeds!) all its goals. We also hope to contribute SIUNITS,[2] developed in part under auspices of the Zoom project library at Fermilab, to the Boost organization [`http://www.boost.org`] to obtain additional critical technical comments. After gaining further experience with SIUNITS, we hope to propose an enhanced version of it to the C++ Standardization Committee for consideration as an extension to the current C++ language standard.

2.2 *Le Système international d'Unités*

Throughout the kingdom there shall be standard measures of wine, ale, and corn.... Weights [also] are to be standardized similarly.
— Magna Carta

Le Système international d'Unités [BIP98] is the recognized international standard for describing measurable quantities and their units.[3] In the United States, writing on behalf of the National Institute of Standards and Technology, Taylor states, "Only units of the SI and those units recognized for use with the SI are used to express the values of quantities" [Tay95, p. v].

SI is founded on seven mutually independent *base quantities* (dimensions): LENGTH, MASS, TIME, ELECTRIC CURRENT, THERMODYNAMIC TEMPERATURE, AMOUNT OF SUBSTANCE, and LUMINOUS INTENSITY. The corresponding *units of measure* for describing amounts of these are specified, respectively, as the METER, KILOGRAM, SECOND, AMPERE, KELVIN, MOLE, and CANDELA.

In addition, SI describes a consistent system for expressing new quantities (*e.g.*, ENERGY) in terms of the seven dimensions. In particular, it designates a list of 22 such *derived quantities* whose amounts are identified via specially named (*e.g.*, JOULE) composites of the base units. Finally, SI defines a list of prefixes for forming units' decimal multiples and sub-multiples.

2.3 SIUNITS basics

The mole is a quantity of substance. The new prefix "guaca" is defined such that one guacamole equals Avocado's Number.
— G. Byrne

At its core, SIUNITS provides, in the form of data types, all base quantities specified by SI. Further, it provides the ability to declare additional data types to represent derived quantities, also as specified. For programming convenience, names of a significant number of derived quantities have been included in SIUNITS. A programmer is free to use, ignore, or augment these as may be convenient. Similarly, all the base and derived units specified by SI are provided in SIUNITS, as are forms of the SI-mandated prefixes. An extensive collection of diverse units from published sources has also been included; these encompass traditional MKS, CGS, U.K., and U.S. units. Many units not in common use, including non-SI units, have also been provided in order to demonstrate both the diversity and the flexibility that SIUNITS facilitates. Programmers can provide their own custom units, too.

A significant collection of constants of nature is also incorporated in SIUNITS. Because any such constant can be used as a unit (*e.g.*, MACH reflects the speed of sound), these constants of nature have been internally combined with traditional units data. As before, a programmer may use, ignore, or augment these. Perhaps most importantly, use of SIUNITS permits to its users only meaningful combinations of units: all incommensurate expressions are diagnosed by the compiler! Figure 1, for example, shows[4] both successful and unsuccessful instantiations.

[2] We have recently become aware of a LaTeX package also named `SIunits`. While the name overlap is unfortunate, this software [Hel01] is designed for text processing, and thus has no functional overlap with our library.

[3] "The Convention of the Metre (*Convention du Mètre*) is a diplomatic treaty ... signed in Paris in 1875.... [It] established a permanent organizational structure for member governments to act in common accord on all matters relating to units of measurement" [BIP00].

[4] Most namespace information is omitted, throughout this paper, to improve clarity of exposition.

```
Mass<>    m = 100 * kilogram;  // ok: kg is a unit of mass
Energy<>  e = m * c * c;       // ok per Prof. Einstein
Length<>  w = m;               // incommensurate; diagnosed
Area<>    a = w * w;           // ok after fixing w
Volume<>  v = 2 * a;           // incommensurate; diagnosed
Time<>    t = 2*min + 49*sec;  // ok; commensurate
```

Figure 1: Successful and unsuccessful instantiations using SIUNITS

```
Length<>  s1 = 3 * cm;  // one side of a right triangle
Length<>  s2 = 4 * in;  // second side
Length<>  h  = sqrt( s1*s1 + s2*s2 );  // hypotenuse
```

Figure 2: Implicitly generating and checking anonymous types

Where necessary, SIUNITS automatically generates and checks implied quantities, such as for intermediate results. Figure 2 illustrates this feature via a straightforward application of the Pythagorean theorem to calculate the hypotenuse of a right triangle. During this calculation, each side's LENGTH will be squared. The compiler will determine that each of these intermediate results is an AREA and will, in turn, check them for commensuration before generating code summing them. Finally, the compiler will determine that the square root of the summed AREA will produce a LENGTH. Because this is commensurate with `h`'s declared type, expressed in the C++ code as `Length<>`, the compiler will translate the initialization without complaint.

Because all SIUNITS' type checking takes place at compile-time and because of extensive inlining, there is typically no run-time performance penalty relative to computation on the underlying representation type (*e.g.*, `double`). Use of certain advanced features, however, may carry run-time costs. For example, expressions that involve mixed *views* (see § 4.1) involve implicit conversions that some compilers may defer to run-time.

3 Implementing quantities

3.1 Early work

One of the main causes of the fall of the Roman Empire was that, lacking zero, they had no way to indicate successful termination of their C programs.
— Robert Firth

The first challenge is to devise a means of representing quantities as data types so that a compiler can deduce a new quantity from an arbitrary combination of existing quantities. Our first approach to this problem, as documented in [Bro98], independently rediscovered the template-based technique described in [BN94]. The key observation is that a unique ordered 7-tuple can encode a dimensionality, associating one constituent with each base quantity specified by SI. Each constituent represents, in this scheme, a *dimensional exponent*, the number of times the corresponding base quantity appears in the dimensionality being encoded. Table 1 gives several examples of quantities and the 7-tuple encodings of their respective dimensionalities.

Early implementations of SIUNITS applied this encoding scheme via seven non-type template arguments, approximately as shown in Figure 3. Modern metaprogramming techniques, however, such as those described in [CE00] and [Ale01], have made several improvements possible.

3.2 The `Ratio<>` template

Arithmetic is being able to count up to twenty without taking off your shoes.
— Mickey Mouse

We developed and introduced into SIUNITS a new template, `Ratio<>` (see Figure 4), as a replacement for the use of the `int` non-type template parameters. This family of types, intended

Quantity	*Encoding*	*Remarks*
SCALAR	0, 0, 0, 0, 0, 0, 0	dimensionless quantity
LENGTH	1, 0, 0, 0, 0, 0, 0	single base quantity
AREA	2, 0, 0, 0, 0, 0, 0	LENGTH squared
VOLUME	3, 0, 0, 0, 0, 0, 0	LENGTH cubed
MASS	0, 1, 0, 0, 0, 0, 0	single base quantity
TIME	0, 0, 1, 0, 0, 0, 0	single base quantity
FREQUENCY ...	0, 0, -1, 0, 0, 0, 0	reciprocal TIME
VELOCITY	1, 0, -1, 0, 0, 0, 0	LENGTH per TIME
ACCELERATION	1, 0, -2, 0, 0, 0, 0	LENGTH per TIME squared
ENERGY	2, 1, -2, 0, 0, 0, 0	consistent with $e = m\,c^2$

Table 1: Representative encodings of quantities' dimensionality

```
template< class Rep
        , int Len, int Mas, int Tim, int Tmp, int Cur, int Amt, int Lum
        >
class Quantity;
```

Figure 3: Declaring quantities in early SIUNITS

to represent compile-time rational numbers, is more general than simple compile-time integral values, permitting us to represent a broader range of dimensional exponents for `Quantity<>` types.

Such an extended range of dimensionalities is highly desirable. As a minor point, the availability of rational powers[5] allowed SIUNITS to dispense with error-checking to ensure, for example, that only even powers were involved when users take any `Quantity`'s square root. Much more significant, however, is the additional computational flexibility that the availability of rational powers affords to SIUNITS' users. For example, the formula $v = \sqrt{e}/\sqrt{m}$ describes a calculation that yields a VELOCITY by taking the square root of an ENERGY and dividing by the square root of a MASS. Since, by definition, $\sqrt{m} \equiv m^{1/2}$, this approach can't be directly applied when users are restricted to integral powers, but can be used under our scheme.

As an implementation detail, `Ratio<>` depends on `Gcd<>`, which computes the greatest common divisor of its template arguments. Shown in Figure 5, this small piece of metaprogramming elegantly implements the classic Euclidean algorithm (extended to signed integers) as a compile-time computation.

`Gcd<>`, in turn, depends on the two supporting templates shown in Figure 6. `Signum<>` computes its template argument's sign, and `Abs<>` computes its template argument's absolute value. These, too, qualify as metaprogramming constructs since each is designed to perform its computation at compile time.

`Ratio<>` is accompanied by a number of additional templates that carry out, at compile-time, the usual arithmetic operations on rational numbers. One example, `RatioAdd<>`, is shown in Figure 7.

3.3 The `List<>` template

Man muß immer generalisieren. (One must always generalize.)
— Carl Gustav Jacob Jacobi

A second improvement over our early `int`-based approach is the use of a `List<>` template. Similar in design to the `Cons<>` template described in [CE00, §10.10.5] and to the `Typelist<>` template found in [Ale01, chapter 3], `List<>` and its helper templates allow us to treat a quantity's dimensionality as a single entity rather than as seven entities. This contrasts with the early

[5] For some applications, even rational powers may seem insufficient. To date, however, no compelling application for this feature has been brought to our attention. It is therefore unclear whether extending our scheme to incorporate floating-point powers has sufficient benefit to warrant the additional complexity, including language support for floating-point nontype template parameters.

```
template< long N, long D=1L >
struct Ratio  {  // rational number N/D
private:
  static  long const  gcd = Signum<D>::ans * Gcd<N,D>::ans;
public:
  static  long const  num = N , den = D;
  typedef  Ratio< num/gcd , den/gcd >  Reduced;

  template< class Into >
  static  Into  cvt()  { return static_cast<Into>(N)
                              / static_cast<Into>(D); }
};  // Ratio<N,D>

template< long D >
struct Ratio<0L,D>  {  // ensure canonical form for Ratio<0L>
  static  long const  num = 0L , den = 1L;
  typedef  Ratio<num,den>  Reduced;

  template< class Into >
  static  Into  cvt()  { return static_cast<Into>(0); }
};  // Ratio<0>
```

Figure 4: Metaprogramming rational numbers

```
template< long M, long N >  struct Gcd  {  // via Euclid's method
private:
  static  long const  m = Abs<M>::ans
  static  long const  n = Abs<N>::ans;
public:
  static  long const  ans = Gcd<n,m%n>::ans;
};  // Gcd<M,N>

template< long M >  struct Gcd<M,0L>  {  // final step of Euclid's method
  static  long const  ans = Abs<M>::ans;
};  // Gcd<M,0>

template<>  struct Gcd<0L,0L>  {  // special case
  static  long const  ans = 1L;
};  // Gcd<0,0>
```

Figure 5: Greatest common divisor computation via metaprogramming

efforts described above, in which the seven-fold repetition of code handling template arguments led to bulk that obscured the underlying logic of the application. We thus achieve the benefits of code factorization, including ease of comprehension, leading to significant reductions in coding, testing, and maintenance effort. Indeed, we reduce our code by very nearly a factor of fourteen this way: each dimensionality has seven components, each a rational number with its own numerator and denominator, each of which would have needed a separate template parameter.

We wrote the `Decompose<>` template (see Figure 8) in anticipation of frequent need to access the individual (`Ratio<>`) components of our `List<>`s. Somewhat surprisingly, we have only rarely found it necessary to do so. Instead, `List<>`'s helper template `ListCombine<>` (see Figure 9) has proven exceptionally useful in the context of SIUNITS.

Central to `ListCombine<>`'s design is the template template parameter named `Op`. Itself taking two template parameters, the `Op` parameter is intended to allow any type computation combining a pair of types. As a specific example, the previously-described `RatioAdd<>` template performs such a type computation. In turn, `ListCombine<>` applies `Op<>` to corresponding items from its other two parameters, each a `List<>`, producing a new `List<>` made up of the results of the pairwise type computations. In this way, for example, a new dimensionality can

```
template< long N >
struct Signum  {  static  long const  ans = (N < 0L)  ?  -1L : +1L; };

template<>
struct Signum<0L>  {  static  long const  ans = 0L; };

template< long N >
struct Abs  {  static  long const  ans = (N < 0L)  ?  -N  :  N;  };
```

Figure 6: Sign and absolute value computations via metaprogramming

```
template< class R1, class R2  >  struct RatioAdd;

template< long N1, long D1, long N2, long D2 >
struct RatioAdd< Ratio<N1,D1>, Ratio<N2,D2> >  {
  typedef  typename Ratio< N1*D2 + N2*D1 , D1*D2 >::Reduced
           Ans;
};  // RatioAdd<R1,R2>
```

Figure 7: Sum of rational numbers via metaprogramming

be obtained, at compile time, from a pair of existing dimensionalities as illustrated in Figure 10. (SIUNITS provides over 400 names for quantities whose dimensionalities are formed in this way.)

4 Commensuration

4.1 Introducing viewpoints

It does not do to leave a live dragon out of your calculations.
— J. R. R. (John Ronald Reuel) Tolkien

Mensuration is intimately affected by the way we regard the universe. While one particular viewpoint certainly predominates, alternate viewpoints are often useful for specialized applications.

For example, to simplify certain computations, high-energy physicists often assume c, the speed of light, to be unity (one). Such an apparently simple (relativistic) viewpoint, however, quickly leads to profound consequences: the usual LENGTH and TIME dimensions must become commensurate! Here is why:

1. Any VELOCITY can be expressed as some scalar multiple of lightspeed c. Since the relativistic $c = 1$, also a scalar, it follows that any relativistic VELOCITYs must be scalar.
2. In order that LENGTH over TIME (the definition of VELOCITY) produce a scalar quantity, these dimensions must cancel when divided, only possible if the dimensions are unified.

```
#define  DimList(a,b,c,d,e,f,g)\
  List<a,List<b,List<c,List<d,List<e,List<f,List<g\
      >      >      >      >      >      >      >

template< class a, class b, class c, class d, class e, class f, class g >
struct Decompose<DimList(a,b,c,d,e,f,g) >  {
  typedef  a  Len;   typedef  c  Tim;   typedef  e  Tmp;   typedef  g  Lum;
  typedef  b  Mas;   typedef  d  Cur;   typedef  f  Amt;
};  // Decompose<DimList()>
```

Figure 8: Accessing a `List<>`'s dimensionality information via metaprogramming

```
template< class L1
        , template<class,class>class Op
        , class L2                      >  struct ListCombine;

template< template<class,class>class Op >
struct ListCombine<EmptyList, Op, EmptyList>  {  typedef  EmptyList  Ans;  };

template< class H1, class T1  // L1 == List<H1,T1>
        , template<class,class>class Op
        , class H2, class T2  // L2 == List<H2,T2>
        >  // assume ListSize<L1>::ans == ListSize<L2>::ans
struct ListCombine< List<H1,T1>, Op, List<H2,T2> >  {
  typedef  List< typename Op<H1,H2>::Ans
               , typename ListCombine<T1,Op,T2>::Ans
               >
           Ans;
};  // ListCombine<L1,Op,L2>
```

Figure 9: Combining two `List<>`s' corresponding types via metaprogramming

```
#define  MakeIntegralDim(a,b,c,d,e,f,g)\
  DimList(Ratio<a>,Ratio<b>,Ratio<c>,Ratio<d>,Ratio<e>,Ratio<f>,Ratio<g> )

typedef  MakeIntegralDim(1,0,0,0,0,0,0)  LengthDim;
typedef  MakeIntegralDim(0,0,1,0,0,0,0)  TimeDim;

typedef  ListCombine<LengthDim,RatioDiv,TimeDim>::Ans  VelocityDim;
```

Figure 10: Applying `ListCombine<>` to form a derived dimensionality

3. In turn, this makes it meaningful, in this context, to sum LENGTHs and TIMEs. Thus, in a relativistic view, LENGTHs are typically measured in LIGHT-SECONDS[6] or similar units.

Further, there are ripple effects. For example, MASS and ENERGY quantities are related by Prof. Einstein's famous equation, $e = mc^2$. If $c = 1$, however, MASS and ENERGY suddenly become commensurate!

To enable such variant uses, SIUNITS supports the concept of a *view*, a self-consistent set of constants and rules for expressing values of assorted quantities. Each view follows the rules laid down by SI, yet is faithful to its underlying assumptions and their consequences.

Five such views (known as the *standard*, *relativistic*, *high-energy physics*, *quantum*, and *natural* views) form an integral part of SIUNITS. Each provides its own self-consistent choices[7] for the units used with the base quantities, and for the rules by which commensuration is determined. Users pay no price for such availability of multiple views; multiple views may be freely mixed within a single compilation unit, and even (although at a small cost for a view conversion) within a single expression. Further, a knowledgeable user may augment SIUNITS with additional views. To avoid working with huge or tiny floating-point numbers, for example, a view can be provided that scales quantities as desired.

4.2 Implementing views

The purpose of computing is insight, not numbers.
— Richard Wesley Hamming

The desire to support multiple views adds several interesting complications to SIUNITS' implementation. To begin, we must fully describe each supported view. Figures 11 and 12 illustrate the metaprogramming encodings we use for the standard and relativistic views, respectively. We

[6] A *light-second* is conventionally defined as the distance that light travels in one second.
[7] A colleague, Mark Fischler, has elucidated the supplied views' mathematical underpinnings in [Fis00].

```
#define SI_calibrate(baseUnit,calibration,tag)\
  static  long double  baseUnit()  { return calibration##L; }\
  static  std::string  baseUnit##Tag()  { return tag; }

struct StdView  {
  template< class Dim >
  struct DimMap  {  typedef  Dim  Ans;  };  // identity map

  SI_calibrate(meter   ,1.0,"m")  SI_calibrate(kilogram,1.0,"kg")
  SI_calibrate(second  ,1.0,"s")  SI_calibrate(ampere  ,1.0,"A")
  SI_calibrate(kelvin  ,1.0,"K")  SI_calibrate(mole    ,1.0,"mol")
  SI_calibrate(candela ,1.0,"cd")

  template< class Dim >
  struct Conversion  {
    static  long double  factor()  { return 1.0L; }
  };  // Conversion<>
};  // StdView
```

Figure 11: Implementing the standard view of mensuration

will briefly discuss each of the three sections constituting these encodings: the `DimMap<>`, the *calibrations*, and the `Conversion<>::factor()` function.

The `typename DimMap<>::Ans` represents a dimensionality. In particular, given as its template argument a dimensionality from the standard view, it produces the equivalent dimensionality in the view being described. Thus, while `DimMap<>` is an identity mapping in the standard view, in the relativistic view it accomplishes the merging (as described in the previous section) of the LENGTH and TIME dimensions.

A view's *calibrations* section consists of fourteen `static` member functions, one pair for each of SI's seven base quantities. Within each such pair, one function provides the view's value for the corresponding base unit. For example, a relativistic METER corresponds to approximately 3.335×10^{-9} LIGHT-SECONDs, while a relativistic KILOGRAM corresponds to approximately 5.609×10^{35} ELECTRON-VOLTs.

The other function within each pair provides the base quantity's default unit label. In the standard view, this label exactly matches SI's specification. In other views, however, the labels are adjusted to the prevailing units. For example, in the relativistic view, MASS is measured in ELECTRON-VOLTs, while the LENGTH unit is unused — it has been merged, after all, with the TIME unit.

Finally, the `Conversion<>::factor()` function, as its name suggests, produces the factor needed to convert values expressed in the standard view to the current view. Conversions from the standard view are performed via multiplication, while conversions to the standard view are done via division. This logic is encapsulated in the `Conversion<>::factor()` function, which provides the appropriate factor to convert a value of a given dimensionality between an arbitrary pair of views. The two views and the dimensionality are supplied via template arguments as shown in Figure 13.

4.3 Checking commensuration

There is no transfer into another kind, like the transfer from length to area and from area to solid.
— Aristotle

The main user interface to quantities in SIUNITS is via templates such as `Mass<>` (see Figure 14). Each of these, in turn, inherits from an instance of the `Anon<>` template whose implementation is outlined in Figure 15. In particular, the `Commensurate<>::use()` construction, so frequently employed, triggers significant compile-time machinery that both checks for commensuration (even across views) and, if needed, converts a value from one view to the corresponding value in another view. Commensuration failure at this point will result in a compilation error as described below.

```
typedef  Ratio<0>  Unused;

struct RelView  {
  template< class Dim > struct DimMap;
  template< class Len, class Mas, class Tim, class Cur, class Tmp, class Amt, class Lum
          >
  struct DimMap<List7(Len,Mas,Tim,Cur,Tmp,Amt,Lum) >  {
  private:
    typedef  typename RatioAdd<Len,Tim>::Ans           LenTim;
  public:
    typedef  List7(Unused,Mas,LenTim,Cur,Tmp,Amt,Lum)  Ans;
  };  // DimMap<>

  SI_calibrate(meter   ,3.33564095198152043e-09,"?")  // light-sec; unused
  SI_calibrate(kilogram,5.60958616694955767e+35,"eV")
  SI_calibrate(second  ,1.0,"s")  SI_calibrate(ampere  ,1.0,"A")
  SI_calibrate(kelvin  ,1.0,"K")  SI_calibrate(mole    ,1.0,"mol")
  SI_calibrate(candela ,1.0,"cd")

  template< class Dim >  struct Conversion;
  template< class Len, class Mas, class Tim, class Cur, class Tmp, class Amt, class Lum
          >
  struct Conversion<List7(Len,Mas,Tim,Cur,Tmp,Amt,Lum) >  {
    static  DefaultRep  factor()  {
      using namespace std;
      return pow( meter   (), Len::template cvt<DefaultRep>() )
           * pow( kilogram(), Mas::template cvt<DefaultRep>() )
           ;
    } // factor()
  };  // Conversion<>
};  // RelView
```

Figure 12: Implementing the relativistic view of mensuration

SIUNITS determines commensuration, at its core, via the metaprogramming code embodied in the `ViewCommensurate<>` template. Shown in Figure 16, this template evaluates, at compile time, the commensuration of two quantities by considering the relationships between the quantities' respective views and dimensionalities.

The two views and the two dimensionalities constitute this template's arguments. Only if the dimensionalities are identical in a common view can the quantities be considered commensurate; this is implemented via the template's specializations. If the views are not identical, the template's general case applies each view's `DimMap<>` to both dimensionalities: if the mapped dimensionalities are identical in either view, then the quantities are considered commensurate.

Putting all this together, we have the final `Commensurate<>` template[8] shown in Figure 17. In brief, this template first evaluates the commensuration of its two template argument quantity types. If commensurate, and in case the two quantity types do not use the identical underlying representation, the template selects the wider of the two representation types. If incommensurate, a type is selected that will give rise to a compiler diagnostic when it is first used: we declare (but deliberately do not define) the `Incommensurate<>` template for this purpose.

All this is accomplished with the help of additional metaprogramming templates:

- an `If<>` template (such as presented in [CE00]) to make general compile-time type selections,
- `Promote<>` and `OpResult<>` templates to determine the types of expressions' results, and
- `Traits<>` templates throughout to enable seamless integration of native C++ types with SIUNITS' `Quantity`-derived types.

We omit the details of these helpers to conserve space.

[8] We intend to recode this template to improve separation of concerns. We envision, for example, that a `CommensurateConcept<>` template in the style of [SL00] will emerge from such a recoding.

```
template< class FromView, class ToView, class Dim >
struct Conversion  {
  typedef  typename ToView  ::template Conversion<Dim>  CvtTo;
  typedef  typename FromView::template Conversion<Dim>  CvtFrom;

  static  long double  factor()  { return CvtTo::factor() / CvtFrom::factor(); }
};  // Conversion<>

template< class View, class Dim >
struct Conversion<View,View,Dim>  {  // identity conversion between same views
  static  long double  factor()  {  return 1.0L; }
};  // Conversion<V,V,D>
```

Figure 13: Converting between arbitrary views

```
typedef  MakeIntegralDim(0,1,0,0,0,0,0)  MassDim;

template< class R=double, class V=StdView >
struct Mass : public Anon<R,V,MassDim>  {
  typedef  Anon<R,V,MassDim>  BaseAnon;
  Mass() : BaseAnon()  {}
  Mass( Mass<R,V> const & orig )
  : BaseAnon(Traits<BaseAnon>::make(Traits<Mass<R,V> >::pure(orig)
           )                        )  {}
  template< class Q >  Mass( Q const & orig ) : BaseAnon(orig)  {}
  template< class Q >  Mass &  operator = ( Q const & rhs )  {
    BaseAnon::operator =( BaseAnon(rhs) ); return *this; }
};
```

Figure 14: A representative user interface to SIUNITS quantities

Finally, the `Commensurate<>` template provides a static member function, `use()`, that is the primary interface to `Commensurate<>`'s facilities. Given a quantity of a binary operation's right-hand type, the function uses the representation type (determined as described in the previous paragraph) to produce that quantity's underlying value, converted as needed (via `Conversion<>::factor()`) to match the view espoused by the binary operation's left-hand type. This `Commensurate<>::use()` function, then, represents the key to SIUNITS' central purposes, assuring commensuration and, simultaneously, providing view independence. It is employed, in SIUNITS, as illustrated in several of `Anon<>`'s member functions (*e.g.*, operators += and *=) shown in Figure 15, and as further illustrated by the nonmember functions presented in Figures 18, 19, and 20.

5 Experience

5.1 C++ compilers

The danger already exists that the mathematicians have made a covenant with the devil to darken the spirit and to confine man in the bonds of Hell.
— St. Augustine of Hippo

Frankly, SIUNITS' coding presented more than the usual programming challenges. The primary barriers were presented by noncompliant compilers. Most of these failed to comply with the C++ standard primarily in the various areas related to template support, especially template template parameters, member templates, and template partial specialization. Finally, uneven support of namespaces and argument-dependent lookup has been problematical.

Secondary issues include compiler performance: virtually all our compilers, independent of platform, tend to take extraordinarily long times to compile template-intensive code when high

```
struct Quantity  {};

template< class R, class V, class D >
class Anon : public Quantity  {
  typedef  Anon<R,V,D>  MyType;

public:
  typedef  R  Rep;  // Underlying representation type, e.g. double
  typedef  V  View; // View, e.g. relativistic
  typedef  D  Dim;  // View-independent dimensionality

  explicit  Anon() : n()  {}  // Default constructor

  Anon( MyType const & q ) : n(q.n)  {}  // Copy constructor

  template< class Q >  // Copy-like constructor
  Anon( Q const & orig ) : n( Commensurate<MyType,Q>::use(orig) )  {}

  template< class Q >  // Alternate numeric value
  Rep  measuredIn( Q const & alt ) const  {
    return n / Commensurate<MyType,Q>::use(alt);  }

  template< class Q >  // Binary += operator
  Anon &  operator += ( Q const & q )  {
    n += Commensurate<MyType,Q>::use(q); return *this;  }

  template< class U >  // Scalar *= operator
  Anon &  operator *= ( U const & u )  {
    n *= Commensurate<Anon<R,V,ScalarDim >,U>::use(u); return *this;  }

private:
  Rep  n;  // Instance data
};  // Anon<R,V,D>
```

Figure 15: An outline of quantities' implementation in contemporary SIUNITS

optimization levels are requested. While we have seen some performance improvements in this area over the last three years, considerably more is still needed.

Compiler diagnostics, too, are in vast need of improvement when templates are involved. We have seen individual messages that don't even fit on a single screen! Fortunately, programming errors related to SIUNITS commensuration checking give rise to diagnostics that typically incorporate the word "Incommensurate" as described in § 4.3. SIUNITS' users find this very helpful.

Overall, SIUNITS has proven to be an excellent stress test for C++ compilers. Based on both language compliance and availability across multiple platforms, our preferred compiler as of this writing is KAI 4.0f [`http://www.kai.com/C_plus_plus/index.html`]. We respectfully but steadfastly decline to incorporate any significant workarounds for manifestly broken or outdated compilers, believing that such activity is an unnecessary and expensive burden[9] that also sends the mistaken message that programmers willingly contort otherwise-compliant code to compensate for compilers' lack of standards compliance.

5.2 A case study

The justification of such a mathematical construct is solely and precisely that it is expected to work.
— John von Neumann

In a study of SIUNITS' utility, we observed a user's progress as he implemented a simple function, of his choice, selected from a standard reference in the high-energy physics community.

[9] The archives of the Boost organization's mailing list bear witness to the oft-heroic efforts put forth by Boost's contributors to make otherwise-conforming code work in noncompliant environments. For whatever reason, issues prompted by Microsoft's Visual C++ 6.0 compiler seem to come up more than those engendered by other environments.

```
template< class V1, class D1, class V2, class D2 >
struct ViewCommensurate  {  // general case handles distinct views
  static  bool const  ans   // try both views' mappings
    =  ViewCommensurate< V1, typename V1::template DimMap<D1>::Ans
                       , V1, typename V1::template DimMap<D2>::Ans
                       >::ans
    || ViewCommensurate< V2, typename V2::template DimMap<D1>::Ans
                       , V2, typename V2::template DimMap<D2>::Ans
                       >::ans;
};  // ViewCommensurate<>

template< class V, class D >
struct ViewCommensurate<V,D,V,D>  {  // same views, same dimensionalities
  static  bool const  ans = true;    // commensurate
};  // ViewCommensurate<V,D,V,D>

template< class V, class D1, class D2 >
struct ViewCommensurate<V,D1,V,D2>  {  // same views, distinct dimensionalities
  static  bool const  ans = false;     // incommensurate
};  // ViewCommensurate<V,D1,V,D2>
```

Figure 16: Metaprogramming to evaluate commensuration, accounting for views

The chosen function computes an electron's final energy after traversing a thickness of a given material; an important part of this requires calculation of a *radiation length*, X_0, according to the following formula reproduced from [M$^+$94, eq. 10.16]:

$$\frac{1}{X_0} = 4\,\alpha\,r_e^2\,\frac{N_A}{A}\,\left\{Z^2\,[L_{rad}\,-\,f(Z)]\,+\,Z\,L'_{rad}\right\}$$

For simplicity, the user decided to ignore the $f(Z)$ term in drafting the function shown in Figure 21 (reformatted and updated to conform to the current version of SIUNITS). He also assumed $Z > 4$ to allow calculating Tsai's constants L_{rad} and L'_{rad} per [M$^+$94, Table 10.2, row "Others"].

To the user's extreme surprise, this brief and seemingly-straightforward function failed to compile due to commensuration errors that SIUNITS reported. In fact, he was initially convinced that his code had uncovered a bug in SIUNITS, rather than the opposite, and only reluctantly proceeded to begin reconsidering his code. To his further surprise, his investigations actually revealed three separate problems, none of which, we believe, would have come to light at compile-time without SIUNITS.

The first error arose due to misleading nomenclature: a *radiation length* is not a LENGTH at all. Rather, a radiation length is an effective thickness, a distance that has been adjusted for a material's density. More careful analysis showed that X_0 should be declared instead as a MASS per AREA, a derived quantity whose corresponding type is known in SIUNITS as an `AreaDensity<>`. This analysis was subsequently confirmed in a later edition of the reference: the clarifying phrase "usually measured in $g\,cm^{-2}$" was added to its updated definition of *radiation length* [G$^+$00, § 23.4.1].

After correcting the declaration, SIUNITS found a problem we consider a transcription error. While the right side of the above formula was correctly coded as written, the left side indicates that it gives X_0's reciprocal, not X_0 directly. A division was inserted to account for the reciprocal.

Finally, a physics error was detected. Likely a consequence of the conceptual error introduced by the unfortunate nomenclature, the user had neglected, in the `return` statement, to adjust the thickness for density. All three of these errors are corrected in the function's revision shown in Figure 22.

In sum, even in this small, relatively simple application, SIUNITS exposed errors due to non-optimal nomenclature (`Length<>`), a transcription oversight (X_0^{-1}), and incorrect physics (a simple oversight). Without SIUNITS, none of these errors would likely have been noticed by the compiler, since all types would probably have been `doubles` or `long doubles`. Run-time testing would have shown wrong results, of course, but their causes would not have been obvious. With SIUNITS, compile-time type errors pinpointed the exact lines in trouble.

```
template< class D1, class D2 >  struct Incommensurate;

template< class Q1, class Q2 >
struct Commensurate  {
private:
  typedef  typename Traits<Q1>::Rep  Rep1;   typedef  typename Traits<Q2>::Rep  Rep2;
  typedef  typename Traits<Q1>::View View1;  typedef  typename Traits<Q2>::View View2;
  typedef  typename Traits<Q1>::Dim  Dim1;   typedef  typename Traits<Q2>::Dim  Dim2;

public:
  static  bool const  ans = ViewCommensurate<View1,Dim1,View2,Dim2>::ans;

  typedef  typename If< ans
                     , typename Promote<Rep1,Rep2>::Ans
                     , Incommensurate<Dim1,Dim2>
                     >::Ans
           Rep;

  static  Rep  use( Q2 q )  {
    return static_cast<Rep>( Traits<Q2>::pure(q)
                           * Conversion<View2,View1,Dim1>::factor()
                           );
  }  // use()
};  // Commensurate<>
```

Figure 17: Metaprogramming to address commensuration, type promotion, and view conversion

```
template< class Q1, class Q2 >
inline  bool  operator < ( Q1 q1, Q2 q2)  {
  return  Commensurate<Q2,Q1>::use(q1)
        <  Commensurate<Q1,Q2>::use(q2);
}  // operator < ()
```

Figure 18: One of SIUNITS' relational operators

6 Summary

NASA's Mars Climate Orbiter was lost ... because engineers failed to [convert] from English units to metric, an embarrassing lapse that sent the $125 million craft fatally close to the Martian surface.
— The Washington Post

In this paper, we have presented several of the metaprogramming implementation techniques underlying SIUNITS: THE LIBRARY OF UNIT-BASED COMPUTATION. After briefly surveying the current state of the art with respect to numeric programming involving units, we have motivated and articulated the fundamental implementation of the `Anon<>` template that implement quantities and their units in a manner consistent with the specifications of *le Système international d'Unités*, the recognized international standard in this area. We also described our basic technique for detecting incommensuration via strong type checking, motivated and presented the concept of different measurement views, and detailed these views' implementations and their application to the determination of commensuration. Finally, we have shared some of our experiences in designing, coding, and applying SIUNITS.

SIUNITS makes productive use of many other template metaprogramming techniques, including most of the techniques surveyed in [Abr01]. We believe that the composite application of all these techniques in the context of C++ has made it feasible and convenient to apply to computer programming, via SIUNITS, the recommended standard of care in scientific computation.

```
template< class Q1, class Q2 >
inline  typename OpResult<Q1,CommensurateOp,Q2>::Ans
operator + ( Q1 q1, Q2 q2)  {
  typedef  typename OpResult<Q1,CommensurateOp,Q2>::Ans  Ans;
  return Traits<Ans>::make( Commensurate<Q2,Q1>::use(q1)
                          + Commensurate<Q1,Q2>::use(q2)
                          );
}  // operator + ()
```

Figure 19: SIUNITS' binary addition operator

```
template< class Q1, class Q2 >
inline  typename OpResult<Q1,MultiplyOp,Q2>::Ans
operator * ( Q1 q1, Q2 q2)  {
  typedef  typename OpResult<Q1,MultiplyOp,Q2>::Ans  Ans;
  return Traits<Ans>::make( Traits<Q1>::pure(q1)
                          * Traits<Q2>::pure(q2)
                          );
}  // operator * ()
```

Figure 20: SIUNITS' binary multiplication operator

Acknowledgments

We must measure what is measurable, and make measurable what is not so.
— Galileo

Several colleagues reviewed early drafts of this paper. It is my sincere pleasure to thank Messrs. I. Dunietz, M. Fischler, J. Kowalkowski, J. Marraffino, and M. Paterno for their valued encouragement, assistance, and friendship. Thanks are also extended to the paper's referees; their constructive remarks were much appreciated.

References

[Abr01] David W. Abrahams. Generic Programming Techniques. `http://www.boost.org/more/generic_programming.html`, 14 March 2001.

[Ale01] Andrei Alexandrescu. *Modern C++ Design*. Addison-Wesley, 2001. ISBN 0-201-70431-5.

[BIP98] Bureau International des Poids et Mesures, Sèvres Cedex, France. *Le Système International d'Unités*, 1998.

[BIP00] Bureau International des Poids et Mesures. *The Convention of the Metre*, 5 June 2000. `http://www.bipm.fr/enus/1_Convention`.

[BN94] John J. Barton and Lee R. Nackman. Example: Dimensional analysis. In *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples*, pages 493–500. Addison-Wesley, 1994. ISBN 0-201-53393-6.

[Bro98] Walter E. Brown. Introduction to the SI Library of Unit-Based Computation. `http://fnalpubs.fnal.gov/archive/1998/conf/Conf-98-328.html`, 2 September 1998. Presented at CHEP (Computers in High-Energy Physics) '98, Chicago, Illinois.

[CE00] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. ISBN 0-201-30977-7.

[Fag92] Mike Fagan. Soft Typing: An Approach to Type Checking for Dynamically Typed Languages. Technical Report 2-184, Rice University, 31 March 1992.

[Fis00] Mark Fischler. World-Views in the SIUNITS Package. Technical Memorandum Fermilab-TM-2123, Fermi National Accelerator Laboratory, Batavia, Illinois, 13 April 2000.

```
Energy<>  finalEnergy( Element<> const & material
                     , Density<> const   dens
                     , Length<>  const   thick
                     , Energy<>  const   initEnergy
                     )  {
  // get the material's properties:
  AtomicWeight<> const  A = material->atomicWeight;
  AtomicNumber<> const  Z = material->atomicNumber;

  // calculate Tsai's constants (assuming Z > 4):
  Number<> const  L_rad  = log( 184.15 / root<3>( Z ) );
  Number<> const  Lp_rad = log( 1194.  / root<3>(Z*Z) );

  // calculate the "radiation length":
  Length<> const  X_0 = 4.0 * alpha * r_e * r_e
                      * N_A / A
                      * ( Z * Z * L_rad + Z * Lp_rad );

  // calculate the final energy:
  return initEnergy
       / exp( thick / X_0 );
}  // finalEnergy()
```

Figure 21: Initial draft of case study code

[G+00] Particle Data Group: D. E. Groom et al. Review of Particle Physics. *The European Physical Journal*, C15(1):1–878, 2000. Tables, listings, reviews, and errata are available online at `http://pdg.lbl.gov`.

[Hel01] Marcel Heldoorn. The `SIunits` package: Support for the International System of Units, v1.25. `http://www.miktex.org`, 21 July 2001.

[HR70] David Halliday and Robert Resnick. *Fundamentals of Physics*. John Wiley & Sons, Inc., 1970. ISBN 471 34430 3.

[M+94] Particle Data Group: L. Montanet et al. Review of Particle Properties. *Physical Review*, D50(3):1173–1826, 1 August 1994.

[SL00] Jeremy Siek and Andrew Lumsdaine. Concept Checking: Binding Parametric Polymorphism in C++. In *First Workshop on C++ Template Programming, Erfurt, Germany*, 10 October 2000. Available online at `http://oonumerics.org/tmpw00/siek.pdf`.

[Tay95] Barry N. Taylor. Guide for the Use of the International System of Units (SI). Special Publication 811, National Institute of Standards and Technology, Gaithersburg, Maryland, April 1995. Available online at `http://physics.nist.gov/Pubs/SP811`.

```
Energy<>  finalEnergy( Element<> const & material
                     , Density<> const   dens
                     , Length<>  const   thick
                     , Energy<>  const   initEnergy
                     )  {
  // get the material's properties:
  AtomicWeight<> const  A = material->atomicWeight;
  AtomicNumber<> const  Z = material->atomicNumber;

  // calculate Tsai's constants (assuming Z > 4):
  Number<> const  L_rad  = log( 184.15 / root<3>( Z ) );
  Number<> const  Lp_rad = log( 1194.  / root<3>(Z*Z) );

  // calculate the "radiation length":
  AreaDensity<> const  X_0 = 1.0 / ( 4.0 * alpha * r_e * r_e
                                   * N_A / A
                                   * ( Z * Z * L_rad + Z * Lp_rad )
                                   );

  // calculate the final energy:
  return initEnergy
       / std::exp( thick * dens / X_0 );
}  // finalEnergy()
```

Figure 22: Revised case study code