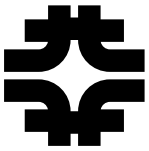

Fermi National Accelerator Laboratory

FERMILAB-Conf-98/328

Introduction to the SI Library of Unit-Based Computation

Walter E. Brown

Fermi National Accelerator Laboratory
P.O. Box 500, Batavia, Illinois 60510

October 1998

Presented at the *International Conference on Computing in High Energy Physics (CHEP '98)*, Chicago, Illinois, August 31-September 4, 1998

Operated by Universities Research Association Inc. under Contract No. DE-AC02-76CH03000 with the United States Department of Energy

Introduction to the SI Library of Unit-Based Computation *

Walter E. Brown†
Fermi National Accelerator Laboratory

September 2, 1998

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

Contents

Credit line: Work supported by the U.S. Department of Energy under contract No. DE-AC02-76CH03000.
Distribution: Approved for public release; further dissemination unlimited.

*An expanded version of this paper is in preparation; please contact the author for details.

†Email: wb@fnal.gov

1 Introduction

1.1 Units-checking and type-safety

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

- "In carrying out any calculation, always remember to attach the proper units to the final result, for the result is meaningless without this...."
- "One way to spot an erroneous equation is to check the dimensions of all its terms...."
- "In any legitimate physical equation the dimensions of all the terms must be the same...."

In modern digital computation, the analogous concept is known as *static 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 accuracy, we paraphrase Halliday and Resnick above (differences **emphasized**):

- In **programming** any calculation, always remember to attach the proper **data type** to the final result, for the result is meaningless without this.
- One way to spot an erroneous **program** is to check the **data types** of all its **objects**.
- In any legitimate **assignment** the **data types** of **both** the **sides** must be the same.

A substantial body of theoretical and applied computing research has been devoted to type-checking. As summarized by Fagan [4, pp. ii, 2], static typing provides "... important feedback to the programmer by detecting a large class of program errors before execution" as well as "... a succinct, intelligible form of program documentation, making programs easier to read and understand." Fagan further notes that "simple conceptual errors" often manifest as type errors, that these are detectable by type-checking software (such as a compiler) used to ensure "... the type documentation is consistent," and (citing [5]) that "[e]xperimental results estimate the number of ... type faults as something between 30% and 80% of all program errors." For these and related reasons, programmers have long been taught to employ and rely on type-checking.

1.2 Prior work

There have been several earlier efforts to attain these benefits. Perhaps the most useful to scientific programmers is currently found in the `Units` subdirectory of the *CLHEP* library. Two files [9, 10] define symbols for units and important constants in common use by the high-energy physics community.

The use of these and similar symbols provides, of course, an important boost to code readability and correctness. However, *CLHEP*'s `Units` files do not express the concepts of the various intended dimensions to which its constants and units belong. All these symbols have type `HepDouble`, typically a synonym for C++'s `double`. Only in a relativistic model where the speed of light is one should it be possible to add meters to seconds; it is otherwise highly unlikely to be a meaningful expression, and thus highly likely to reflect a conceptual and attendant computational error.

1.3 Current programming practice

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 standard of care recommended above seems only rarely applied to numeric quantities. Informal inspection of contemporary code samples revealed that, in numeric programming, programmers make heavy, near-exclusive, use of a language's native numeric types (*e.g.*, `double`). A search of representative on-line archives did not even find any queries on the subject, suggesting little or no interest in, or knowledge of, alternatives.

In light of prior software art, this finding is not surprising. However, such overly-general practice is a common source of errors: it fails to distinguish the diverse intentions (*e.g.*, distances, masses, energies, momenta, *etc.*) that any such purely numeric value can represent. Were such intentions routinely expressed in our computer programs, application of contemporary compiler technology would routinely provide us the benefits of type-safety in our numeric computations as a by-product of the translation process. This benefit is consistent with the unit-based approach so strongly advised by [6], among many others.

1.4 Scope of current project

To address current deplorable 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, thus obtaining 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 without run-time overhead (*i.e.*, at no run-time cost in time or in space).

More specifically, we sought

1. application of current software technology to numeric physical concepts,
2. convenience of expression in such application,
3. general utility rooted in existing standards,
4. use of nomenclature from our problem domain, and
5. no attendant performance penalties!

The present project, known as *The* SI LIBRARY *of Unit-based Computation*, has succeeded in addressing these requirements. The resulting software module (known hereinafter as the SI LIBRARY or, simply, the LIBRARY) meets (and, in many respects, greatly exceeds!) all its goals and is intended for contribution (for non-commercial use) to the FPCLTF ("Zoom") project library at Fermilab.

2 Underpinnings and Basic Features

2.1 International Standard of Units

Le Système international d'Unités (SI) [2] is the recognized international standard for describing measurable quantities and their units. SI specifies seven mutually independent base quantities (dimensions): *length*, *mass*, *time*, *electric current*, *thermodynamic temperature*, *amount of substance*, and *luminous intensity*. The base units 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 dimensions (*e.g.*, energy) in terms of the seven base dimensions. In particular, it includes a list of 21 such derived dimensions whose amounts are described in specially-named composites (*e.g.*, joule) of the base units. Finally, SI provides a list of prefixes for forming units' decimal multiples and sub-multiples.

2.2 Library basics

At its core, *The* SI LIBRARY *of Unit-based Computation* provides, in the form of data types, all the base dimensions specified by SI. Further, it provides the ability to declare additional data types to represent derived dimensions, also as specified.

For programming convenience, a very significant number of derived dimensions have been included in the LIBRARY. In particular, virtually all the dimensions described by Horvath [7] and by Pennycuick [11] are provided. A programmer is free to use, ignore, or add to these as may be convenient for the application at hand.

Similarly, all the base and composite units specified by SI are provided in the LIBRARY, as are forms of the SI-mandated prefixes. An extensive collection of diverse units from published sources (*e.g.*, [7, 11, 3, 9, 10]) have also been included in the module; these encompass traditional MKS, CGS, UK, and US units. Many units not in common use have also been provided in order to demonstrate both the diversity and the flexibility that the LIBRARY enables.

A significant collection of constants of nature is also incorporated in the SI LIBRARY. Because any such constant can be used as a unit (*e.g.*, Mach is based on the speed of sound), these constants of nature have been internally combined with traditional units data; many of the same published sources were used to furnish their values. As before, a programmer may use, ignore, or extend this list.

The SI LIBRARY's infrastructure is furnished to a user program in the form of header files that make known all the LIBRARY-defined data types, units, and constants of nature. All these identifiers are declared within a C++ `namespace` so as to avoid introducing extraneous symbols.

3 Annotated Example

The following sample code first illustrates the fundamentals of instantiating SI LIBRARY objects in connection with SI's basic `Length` dimension. Note that, in the absence of any explicit units, the appropriate base unit (or combination of base units) will be assumed. Further note that any mismatch between units and dimensions will give rise to a compile-time type error.

Part two illustrates computation and output with `Length` and `Length`-related dimensions. Note in particular that the units in which values are displayed may be changed dynamically at will; however, no such change affects the internal representation of the value. While a change in the display of any of the seven SI base dimensions may impact the subsequent display of derived dimensions, no change to the display of a derived dimension will affect the display of any other dimension.

```
#include "SIunits/stdModel.h"
#include <iostream>

int main()  {

  // Preparation:
  using namespace si;     // make most si:: symbols available
```

```
  using namespace si::abbreviations;
  using namespace std;         // make all std:: symbols available

  // -----  Part 1: illustrate instantiation/initialization  -----

  // Successful instantiations:
  Length d;               // initialized to 0 meters by default
  Length d2( 2.5 );       // 2.5 meters; same as 2.5 * m
  Length d3( 1.2 * cm );  // 1.2 centimeters; recorded as .012 * m
  Length d4( 5 * d3 );    // equivalent to 6.0 centimeters
  Length d5( 1.23 * pico_ * meter ); // 1.23e-12 * m
  Length d6( d2 + d3 );   // all dimensions match

  // Bad instantiation attempts; all detected at compile-time:
  Length d7( d2 * d3 );   // oops: an Area can't initialize a Length
  Length d8 = 3.5;        // oops: 3.5 is not a Length
  Length d9;              // so far so good, ...
    d9 = 3.5;             // ... oops: 3.5 is still not a Length

  // -----  Part 2: illustrate computation and output  -----

  // Display with default labels:
  cout << inch << endl;        // display "0.0254 m"

  // Prefer centimeter labels from now on:
  Length::showAs( cm, "cm" );  // set default display units
  cout << inch << endl;        // display "2.54 cm"

  // Compute/display:
  Length len( 2*cm );
  Width  wid( 3*cm );          // Width is a synonym for Length
  Area   a( len * wid );
  cout << a << endl;           // display "6 cm^2"

  // Prefer to label Area in square meters:
  Area::showAs( m*m, "m2" );   // set default display units
  cout << a << endl;           // display "0.0006 m2"
  cout << a*4 << endl;         // display "0.0024 m2"

  // But a volume reverts:
  cout << a * 4*cm << endl;    // display "24 cm^3"

  return 0;

} // main()
```

4 Additional Features

4.1 Choice of data representation

By default, the LIBRARY employs the C++ native `double` data type as its underlying representation. Recognizing, however, that different programming projects may require different underlying data representations (*e.g.*, `float`, `long double`, `complex<float>`, *etc.*),

C++'s `template` mechanism was used throughout this LIBRARY's implementation. This allows a knowledgeable programmer to specify the data representation desired. The memory requirement for each object of an SI LIBRARY type is exactly the same memory amount that would be required for an object of the underlying native type.

4.2 No run-time overhead

Because all type-checking is handled at compile-time and because of heavy use of inlining, there is no run-time computational overhead beyond the time taken for the necessary arithmetic. Thus, use of the SI LIBRARY incurs no performance penalty relative to computation on the native type.

4.3 Choice of calibration

Because different users work with values of radically different magnitudes, the SI LIBRARY allows for certain user calibration. By default, each of the seven base units (meter, kilogram, second, *etc.*) is calibrated to a value of one. However, for users who customarily work, say, in microns and nanoseconds, it is possible to calibrate the LIBRARY to treat these units as the base units instead. Such calibration, however, can only take place at the time the LIBRARY is built; dynamic recalibration is neither currently possible nor envisioned as a future enhancement of the LIBRARY.

4.4 Choice of model

Finally, different programming projects may use different models of the universe. For example, to simplify certain computations, high-energy physics calculations often assume the speed of light to be one. Such assumptions are also possible within the SI LIBRARY and all consequences of such assumptions (*e.g.*, the merging of the `Length` and `Time` dimensions) can be accurately modeled.

The LIBRARY is supplied with five models; these are known, respectively, as the *standard*, *relativistic*, *high-energy physics*, *quantum*, and *natural* models. The following table provides some details as to each model's characteristics; all models use the *mole* as the unit of `AmountOfSubstance` and the *candela* as the unit of `LuminousIntensity`:

Model	Defining Characteristics	Default Units
stdModel	per Système international	m (`Length`); kg (`Mass`); s (`Time`); A (`Current`); K (`Temperature`)
relModel	$c = 1$	s (`Length`, `Time`); eV (`Mass`); A (`Current`); K (`Temperature`)
hepModel	$c = k = e^+ = 1$	ns (`Length`, `Time`); GeV (`Mass`, `Temperature`); e^+ (Charge)
qtmModel	$c = k = \hbar = 1$	inverse GeV (`Length`, `Time`); GeV (`Mass`, `Temperature`, `Current`)
natModel	$c = k = \hbar = G = 1$	numeric only (`Length`, `Time`, `Mass`, `Temperature`, `Current`)

An application program selects the desired model via an `#include "..."` directive. A programmer pays no price for the availability of multiple models; only the code associated with the chosen model need be linked with application code. Additional models may, of course, be added to the LIBRARY by a knowledgeable programmer.

The LIBRARY's five models are designed to provide progressively more restrictive viewpoints. Such "forward compatibility" in models is obtained via recompilation using a header from any more restrictive model, and no other source change. Thus, a user may freely develop code under one model, and later elect to recompile and run with any more restrictive model.

5 Concluding Remarks

This paper has presented the concept of type-checking as the computer analog of manual calculation with units. The benefits of type-checking have been set forth, together with empirical evidence that programmers do not today apply type-checking to its fullest potential in numeric computation.

The thrust of the paper has been the general description of *The* SI LIBRARY *of Unit-based Computation*, a software subsystem intended to apply the documented benefits of type-checking to numeric computation with no run-time overhead in either time or space. Convenience of use, economy of application, and ease of extensibility were primary objectives of the project. Major features of the SI LIBRARY have been discussed and illustrated.

We conclude that the SI LIBRARY software project has met each of its objectives, and has exceeded many of them. To quote sample user reaction to date, "[the SI LIBRARY] is the first really good reason I've seen to switch from Fortran to C++."

6 Acknowledgements

In addition to his general support as chair of the Fermi Physics Class Library Task Force, Mark Fischler first suggested applying the core of the SI LIBRARY to additional models. He also defined the desired properties of the advanced physics models and assisted materially in validating the LIBRARY's behavior in these models. Isi Dunietz also assisted with some of this validation and with proofing.

7 Addendum

Several weeks after this paper was prepared and delivered at CHEP'98, a colleague pointed out the 1994 work of Barton and Nackman [1]. Preliminary analysis of their chapter "Algebraic Structure Categories" suggests some overlap with the present work, which independently discovered some of the same techniques. While Barton and Nackman term their code "advanced examples," their efforts should not be overlooked.

References

[1] Barton, John J. and Lee R. Nackman. *Scientific and Engineering C++: An Introduction with Advanced Techniques and Examples.* Addison-Wesley, 1994. ISBN 0-201-53393-6.

[2] Bureau International des Poids et Mesures. "Le Système international d'Unités." Sèvres Cedex, France, 1998.

[3] Caso, Carlo, *et al.*"Review of Particle Physics." The European Physical Journal, C3 (1998) pp. 69-70, 73.

[4] Fagan, Mike. "Soft Typing: An Approach to Type Checking for Dynamically Typed Languages." Technical Report 92-184, Rice University, March 31, 1992.

[5] Gannon, John D. "An Experimental Evaluation of Data Type Conventions." Communications of the ACM, 20:8 (August, 1977), pp. 584-595.

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

[7] Horvath, Ari L. *Conversion Tables of Units in Science & Engineering.* The Macmillan Press Ltd., 1986. ISBN 0-444-01150-1.

[8] International Standards Organization, JTC1/SC22. *Programming Languages – C++*. ISO/IEC FDIS 14882, 1998.

[9] Maire, Michel and Evgueni Tcherniaev. File: `Units/PhysicalConstants.h`. In *CLHEP* v1.3, CERN, 1998.

[10] Maire, Michel and Evgueni Tcherniaev. File: `Units/SystemOfUnits.h`. In *CLHEP* v1.3, CERN, 1998.

[11] Pennycuick, Colin J. *Conversion Factors: S. I. Units and Many Others*. Univ. of Chicago Press, 1988. ISBN 0-226-65507-5.