YRHI-98-08

Comments on the Use of CLHEP for STAR

STAR NOTE 329

Yale Relativistic Heavy Ion Group

Brian Lasiuk *and* Thomas Ullrich

Department of Physics
Yale University

Brian.Lasiuk@Yale.edu

Thomas Ullrich@Yale.edu

Abstract

A brief description of the functionality of *A Class Library for High Energy Physics* (CLHEP) is described highlighting the differences and similarities with the Standard C++ Library and the Standard Template Library (STL) as well as some possible extensions that exist. Some limitations and complications that will arise as a result of the use of CLHEP are explored.

1 Introduction

Before any substantial code writing is done in C++ for a future STAR class library, decisions must be made regarding what class libraries should and should not be used. The choices made must not limit the programmers from the outset nor impose expensive compatibility problems, however it is important to incorporate useful work that already exists, and use lessons from past and on going collaborations; the most important of these are of course GEANT4 [1], BaBar [2], and those who pioneered the *Class Library for High Energy Physics* (CLHEP) [3].

CLHEP is the oldest effort to generate a standardized C++ class library for the High Energy Physics community much in the spirit of CERNLIB [4], but with more modest goals. Its foundations were laid in 1992 at a conference for Computing High Energy Physics (CHEP 92). It was written before anything like the Standard Template Library (STL) [5] existed and as such some of its functionality is now duplicated. Except for the very specific physics oriented classes, a lot of the work is more efficiently implemented and standardized in the STL. As such the question must be asked which classes are convenient to use and which are better left unused.

2 Introduction to CLHEP

Both the GEANT4 and BABAR collaborations use and contribute to the CLHEP library, so it is a contemporary and evolving class library. It consists of 11 different class **categories** which are listed below with a brief explanation:[1]

- **config** contains several macros as well as `typedef`s of the basic data types.
- **Units** defines a consistent system of SI units as well as some important physical constants.
- **Hist** currently a HBOOK wrapper.
- **Random** provides 5 random engines and 5 random distributions from which pseudo-random numbers can be generated.
- **Vector** defines a `ThreeVector` and `LorentzVector` as well as associated operations and rotations.
- **Matrix** provides matrix operations.
- **Geometry** definitions of geometric shapes that are used in GEANT4. It is dependent on the `ThreeVector` class.
- **Alist** templated lists that are arrays with added functionality such as insertion and removal of entries at any position within the list and sorting capabilities.
- **Combination** is a combinatoric engine that given a list of objects (**Alist**) produces unique combinations or subsets of objects.

[1] A class browser for CLHEP is located at [6].

- **String** contains string handling capabilities as well as a command line parser.

Although there are some categories that can be seen to be immediately replaceable by the STL such as the **String** and **Alist** classes, there are some specific HEP classes that can be used which to ensure compatibility with other software projects outside the STAR collaboration. For example the **Units** category consists of two header files which are both utilized in GEANT4. `PhysicalConstants.h` provides numerical definitions of constants (like h, $\hbar$, π ...) and `SystemOfUnits.h` provides a consistent set of SI units. This is a very desirable feature to avoid ambiguities in calculations and increases the portability of code significantly. The base units are defined as: 'mm', 'ns', 'MeV', 'e', '°K', 'mole', 'radian', and 'steradian'. Derived units can of course be used as well as conversions to other units.[2] For example:

```
static const HepDouble mm    = 1;
static const HepDouble mm2   = mm*mm;
static const HepDouble m     = 1000.*mm;
static const HepDouble ns    = 1.;
static const HepDouble  s    = 1.e+9*ns;
static const HepDouble kg    = joule*s*s/(m*m);
static const HepDouble tesla = volt*s/m2;
```

Here HepDouble is defined in **config** by: "`typedef double HepDouble;`".

The only penalty in using these classes is the rather inelegant way in which this has been implemented[3]. Some examples are:

- Underscores have been used extensively in the names:

  ```
  static const HepDouble electron_mass_c2 = 0.51099906 * MeV;
  ```

- Some constants are defined twice and/or are already defined in system header files. For example `M_PI` is defined in **config** (Pi in **Units**) but also in `<math.h>`. In some cases this is a necessity to guarantee compatibility with vendors which do not provide these constants (i.e. `M_PI` is not defined in *Visual C++* from Microsoft).

- Single letter characters have been used for some units like:

  s for second.

  m for meter.

While the first two points are more aesthetic arguments, the last introduces a real possibility of creating unintended behaviour in code that is very difficult to trace. A somewhat contrived example illustrates the point. In the header file `SystemOfUnits.h` the meter is defined as the base unit by the single character `m` as:

```
static const double m = 1000*mm;
```

[2] see appendix A for details.

[3] CLHEP does not follow the GEANT4 coding guidelines

Since it is in the global name space, m has file scope and is valid throughout the file. If in the file, a loop is defined:

```
double length;
for(int m=0; m< MAXNUMBER; m++) {
    length = 10*m;              // length = 0 for the first iteration
    length = 10*::m;            // length = 10 meters
    ⋮
```

The definition of m as an integer is completely legal but has scope only within the loop. In order to make sure the definition of meter is used, one must be careful to specify the scope or which m is really intended. The first attempt to define a `length` in the above example uses the integer definition of m. The scope resolution operator "::" is required in order to utilize the intended definition. In order to reduce the ambiguity there is the possibility of using a "scope" class (like `numeric_limits` from the Standard C++ library) to define the **Units**. However, ANSI allows only integral types to be constants in class definitions. Therefore a definition for m (or any other unit) would have to be constructed as a member function like:

```
class systemOfUnits {
public:
    static inline float mm() {return (1);}
    static inline float m() {return (1000*mm());}
    ⋮
```

Although this allows the use of the single character m to define the meter, specification of the definition as a member function requires the use of the brackets "()" and this reduces the code readability. For example:

```
length = 10*systemOfUnits::m();
```

This is clear in meaning however, it is not clean syntax and becomes extremely verbose, especially when longer derived units are used.

These are rather inappropriate solutions to hide the real problem of the non-descriptive choice of the symbol to represent *meter* and the predisposition that simple single character names are used for loop counters and temporary stores. This pitfall could be easily avoided with the use of more descriptive and perhaps more appropriate names for definitions in the header file such as:

```
static const double meter = 1;
```

The code could then becomes much less ambiguous and much more readable:

```
length = 10*meter;
```

with greatly reduced worry of mistaking the scope of `meter`. However, this solution would mean discarding the CLHEP header files.

The category **Random** is a fully functional random number generating engine with its own seed generator as well as a choice of various distributions from which numbers can be generated; i.e. flat, Exponential, Gaussian, Lorentzian, ... This is highly desirable to incorporate into a STAR class library. The only short coming is the use of several `extern` statements in some of the header files.

The category **Hist** is simply an HBOOK wrapper so that one can call such routines from a C++ program. This, or a reasonable facsimile, is a necessary evil until it is known what will replace the HBOOK package. The GEANT4 collaboration will no doubt take the lead in this matter and it would be wise to follow their initiative in this manner.

The category **Geometry** contains definition of 3-dimensional shapes and is taken from GEANT4. One possible use would be in the STAR geometry database. It is dependent on the classes defined in **Vector** which unfortunately posses problems of its own.

Vector contains `ThreeVector.h`, `LorentzVector.h` as well as rotation classes. As such they are also closely related to the classes contained in **Matrix** (see below for a more detailed discussion of **Matrix**). In principle these classes are indispensable in a class library in HEP, however there is again a problem in the implementation of these classes. Most significantly is that these are not template classes and are only defined for `double` precision numbers. If high precision is not required, memory usage can be reduced by a factor of two with a change to single precision numbers. This is important considering the large data volume expected and may become an even stronger issue with the eventual migration to 64-bit hardware. On a positive note, the functionality of these classes by themselves appears complete and very user friendly. For example there is a very nice feature of the `LorentzVector` which allows access to the data members in coordinate space: x(), y(), z(), t(), or momentum space: px(), py(), pz(), e(). However a major shortcoming is the relation between the **Vector** and **Matrix** classes. The `ThreeVector` and `LorentzVector` classes are not related via inheritance which means that there is some code redundancy between the two which is unnecessary. From a Heavy-Ion Physics point of view it would be highly desirable to add essential methods to access rapidity (from `LorentzVector`) and pseudorapidity (from `LorentzVector`).

The **Matrix** category, which was briefly mentioned above in its relation to the **Vector** classes is a collection of four different matrix types:

- `GenMatrix.h` defines a purely virtual class that defines the functionality of the class.
- `Vector.h` defines single column matrices for use in rotation
- `DiagMatrix.h` defines diagonal matrices
- `SymMatrix.h` defines symmetric matrices
- `Matrix.h` defines a general matrix with no special symmetry properties.

Although this was done with the intention of maximizing the speed of calculations by taking advantage of certain simplifications with specific matrix properties, the explosion in the amount of code required makes this questionable. In physics applications one utilizes matrices for a simple, and usually one time rotation, where there is generally little concern for every possible type of matrix.

A more general problem is the relation between **Vector** and **Matrix**. A vector is a special case of a single column (or row) matrix and is defined twice in CLHEP. This is done in order to implement the rotation operators. A more careful design of these classes could be made to avoid redundant code.

Although the above classes could be used with little or slight effort, there is no reason to utilize the **String** and **Alist** classes as their functionality is now handled much more efficiently within the STL:

- *<string>* provides string handling in the STL and *<Args.h>*[4] allows the parsing of command line arguments.
- *<vector>* and *<deque>* allow for implementations of templated ordered lists and *<list>* for double linked lists.

Combination is a class that could be used in generating random noise ensembles, however its strong dependence on the **Alist** class does not make it attractive to incorporate.

3 Commercial Alternatives

It is also noteworthy that there are extremely powerful extensions in commercially available and standardized class libraries that posses significant capabilities beyond both CLHEP and STL. For the large majority of programming tasks where object-oriented techniques are the preferred approach products such as Rogue Wave's *Tools.h++* [9] Library which encapsulates the Standard C++ Library with a familiar object-oriented interface, which provides the power and the advantages of object-orientation. It provides several classes that could find immediate applications with physics analysis code:

- Numerous generic and templated collection classes to store objects by value or pointer. Special collection classes are able to handle arrays sizes beyond memory limits by introducing its own optimized disk swap mechanism.
- `BitVector`[5] is a dynamically allocatable vector that can handle quality control and status flags.
- `Bitstream` handles binary data I/O in a form as convenient as a simple iostream.
- `Timer` is a "stop-watch" class that allows the timing of CPU usage in segments of code or complete programs.
- `Tokenizer` and `Regexp` are powerful string parsers which could be used in data base queries as well as command line parsing and class browsers.
- `HashDictionary` could be adapted for fast queries of calibration constants.

[4] Unfortunately *<Args.h>* is not provided by every vendor and is platform specific. However parsers can easily be constructed from the `string` classes of the STL.

[5] A similar class `bitset` is contained in the Standard C++ Library but is **not** dynamically allocatable.

The advantage of using a commercial package is that it is optimized and efficient code that is supported on most all major platforms of interest at RHIC. The classes that are provided are in general templated and capable of object manipulation by *Pointer*, by *Value*, or by *Reference*. Although the utilities listed above are very useful, a very convincing argument for incorporating *Tools.h++* into the STAR library is the fact that the memory management for pointer manipulation is done by two member functions; usually one member like `push_back(T*)` for allocation and `clearAndDestroy()` for deallocation. This greatly simplifies the code writing by alleviating the responsibility of the programmer to make sure the correct number of elements has been looped over and deleted.

Rogue-Wave also provided a math based library, *math.h++*[10]. This library provides functionality quite beyond that of CLHEP in vector and matrix classes as well as random number generators. There also exists a statistics class which also does basic fitting and a signal processing class that allows calculations of Fourier Transforms. However, it does not begin to approach the functionality of CERNLIB and it does come with a steep cost.

4 Complications with the Use of CLHEP

Although there are some very desirable features in CLHEP there are also some serious problems which may compromise our capabilities, many of which have been discovered by the GEANT4 collaboration. As such an important question we have to investigate is whether or not to use all, some, or none of it. A significant problem is that the functions **abs**, **min**, and **max** are defined as a template function with the return type the same as the argument type. The problem lies in the fact that these functions are also defined in the Standard C++ Library (and STL) and the definitions may conflict. In order to get around this, there is a necessity to define these functions only when the STL is not available, and this is handled by a parsing script which is an inconvenient and unnecessary overhead. For example:

```
#ifndef CLHEP_MAX_MIN_DEFINED
#define CLHEP_MAX_MIN_DEFINED
template <class T>
inline const T& min(const T& a, const T& b) {
    const T& retval = b < a ?  b :  a;
    return retval;
}
#endif
```

where `CLHEP_MAX_MIN_DEFINED` is defined at compile time.

This is not an appropriate solution because code readability suffers as the user does not know which instance of the function is being used. Furthermore the problem becomes more complex when code is being mixed and one wants to use both the code from the STL *and* CLHEP in the same scope. This is troublesome because software is generally separated into packages and there is really no way to tell whether some package uses CLHEP or not. Thus, *every single package* must include some standard header file before including anything from any other package if we want to avoid problems. This is definitely not pretty!

The problem is more serious in the case of **abs**. It is a template function defined as:

```
template <class T>
inline T abs(const T& a) {
    return a < 0 ?  -a :  a;
}
```

If one defines a reasonable function `abs()` that returns the magnitude of a vector via:

```
double abs(ThreeVector);
```

the compiler would not be able to handle this because the ANSI standard does not guarantee to recognize overloading by return type; only by argument type![6]

It is interesting is that no call to **abs**, **min**, or **max** is used in CLHEP. As such, it would compile and function quite happily without these definitions, so there is no reason to use them except for the fact that we would be defining our own private version of CLHEP which is not desirable if we want to keep compatibility with other systems. This is a problem that all collaborations have to face that use CLHEP. It seems that in the near future, there will be a renaming within CLHEP such that the convention will be `CLHEP_abs()` or something comparable. Again, we should follow the initiative of the GEANT4 people.

Another complication we will have is the prefixes on the data types. It has been suggested in the *STAR C++ Coding Guidelines* [8] that the prefix **St** should begin each type;
i.e. `typedef StDouble double`. This is standard among most all C and C++ projects. The problem arises when one wants to incorporate several packages or class libraries. For example, say we have GEANT4, CLHEP, and a STAR based C++ class library. Then we have: G4Double, HepDouble, StDouble, as well as the native double which specifies a double precision value.[7] This is more of a minor annoyance as it can be solved with a global type definition file where all the data types can be defined, but it is additional overhead.

In regards to the specific data types CLHEP has allowed for the possibility that the `bool` type does not exist on some platforms. At this time this should not be a problem with any of the STAR sanctioned platforms, or any platform with a reasonable compiler, and as such is redundant, unnecessary code.

5 Conclusions

It seems obvious that if no STL existed, CLHEP provides an attractive foundation to build a class library around. However it is not clear that its functionality, in comparison to the current STL and some of its extensions is sufficient reason to tolerate its shortcomings.

As of December 1997, the official policy within the GEANT4 collaboration is:

> ... recommend you to use only Vector, Geometry, Random, Matrix and Units modules. Very probably the other modules will be removed or completely rewritten in future. [7]

[6] An exception to this is the Borland C++ compiler.

[7] There are also similar definitions from ODMG compliant databases. For example Objectivity will introduce a data type: `ooDouble`.

It is feasible to use CLHEP in the context of the STAR C++ Class Library and probably very desirable from the point of view of keeping compatibility with other HEP experiment libraries. There are some packages that are very important in physics analysis software that will not be supplied by the STL. These include the vector classes, system of units, and physical constants. In order to keep compatibility with other members in the physics community it is very desirable to standardize these and this is a role that CLHEP can provide. Although we can make use of a subset of CLHEP it seems very clear that no feature of CLHEP should be retained that would limit or reduce the flexibility or functionality of the STL. That being said, it is also important to make a compelling case before CLHEP is discarded. The question to ask of these is if they meet the need of the STAR collaboration.

With the expected data volume, a templated `ThreeVector` and `LorentzVector` class is very desirable and we propose to incorporate this along with other member functions that give useful kinematic quantities like rapidity, pseudo-rapidity, and transverse mass. It is also possible to reduce the number of classes by incorporating **Matrix** in a more clever manner and with a closer relationship to the **Vector** classes. The **Units** category is something STAR can implement, however care is required in respecting scope for ambiguous variable names. This is still open to discussion. There is no compelling reason to discard or alter the HBOOK wrapper **Hist** the **Geometry** or **Random** number generator classes and we can easily include them into the STAR class library. There is, however, no need for categories that are much better represented in the STL, most notably **Alist**, **String**, and **Combination** and we propose to remove these along with the template functions **min**, **max**, and **abs**.

References

[1] http://wwwcn.cern.ch/asd/geant/geant4.html

[2] http://www.slac.stanford.edu/BFROOT/doc/www/Computing.html

[3] http://wwwcn1.cern.ch/asd/lhc++/clhep/index.html

[4] http://wwwinfo.cern.ch/asd/index.html

[5] for example see: http://www.roguewave.com/products/abbrevcr/stdref.html

[6] see: http://star.physics.yale.edu/TpcSimulator/classBrowser/browser/classIndex.html

[7] E. Chernyaev, LHC++ listserver, Dec 8, 1997.

[8] http://star.physics.yale.edu/TpcSimulator/coding-guidelines.html

[9] http://www.roguewave.com/products/abbrevcr/tlsref.html

[10] http://www.roguewave.com/products/math/mbrochr2.html

A Units

This is the `SystemOfUnits.h` header file from CLHEP which defines a consistent set of SI units.

```
// -*- C++ -*-
// $Id: SystemOfUnits.h,v 1.6 1997/12/05 15:29:52 evc Exp $
// ----------------------------------------------------------------------
// HEP coherent system of Units
//
// This file has been provided by Geant4 (simulation toolkit for HEP).
//
// The basic units are :
//              millimeter  (mm)
//              nanosecond  (ns)
//              Mega electron Volt  (MeV)
//              positron charge (eplus)
//              degree Kelvin
//              the amount of substance (mole)
//              radian  (rad)
//              steradian (st)
//
// Below is a non exhaustive list of derived and pratical units
// (i.e. mostly the SI units).
// You can add your own units.
//
// The SI numerical value of the positron charge is defined here,
// as it is needed for conversion factor : positron charge = e_SI (coulomb)
//
// The others physical constants are defined in the header file :
//                      PhysicalConstants.h
//
// Author: M.Maire
//
// History:
//
// 06.02.96   Created.
// 28.03.96   Added miscellaneous constants.
// 05.12.97   E.Tcherniaev: Redefined pascal (to avoid warnings on WinNT)

#ifndef HEP_SYSTEM_OF_UNITS_H
#define HEP_SYSTEM_OF_UNITS_H

#include "CLHEP/config/CLHEP.h"

//
// Length [L]
//
static const HepDouble mm  = 1.;                         // millimeter
static const HepDouble mm2 = mm*mm;
static const HepDouble mm3 = mm*mm*mm;

static const HepDouble cm  = 10.*mm;                     // centimeter
static const HepDouble cm2 = cm*cm;
static const HepDouble cm3 = cm*cm*cm;
```

```
static const HepDouble m  = 1000.*mm;                    // meter
static const HepDouble m2 = m*m;
static const HepDouble m3 = m*m*m;

static const HepDouble km = 1000.*m;                      // kilometer
static const HepDouble km2 = km*km;
static const HepDouble km3 = km*km*km;

static const HepDouble microm = 1.e-6*m;                 // micro meter
static const HepDouble  nanom = 1.e-9*m;
static const HepDouble  fermi = 1.e-15*m;

static const HepDouble      barn = 1.e-28*m2;
static const HepDouble millibarn = 1.e-3*barn;
static const HepDouble microbarn = 1.e-6*barn;
static const HepDouble  nanobarn = 1.e-9*barn;

//
// Angle
//
static const HepDouble  rad = 1.;                        // radian
static const HepDouble mrad = 1.e-3*rad;                 // milliradian
static const HepDouble  deg = (3.14159265358979323846/180.0)*rad;

static const HepDouble   st = 1.;                        // steradian

//
// Time [T]
//
static const HepDouble ns = 1.;                          // nano second
static const HepDouble  s = 1.e+9*ns;
static const HepDouble ms = 1.e-3*s;

static const HepDouble  Hz = 1./s;                       // Hertz
static const HepDouble kHz = 1.e+3*Hz;
static const HepDouble MHz = 1.e+6*Hz;

//
// Electric charge [Q]
//
static const HepDouble eplus = 1. ;              // positron charge
static const HepDouble e_SI  = 1.60217733e-19;  // positron charge in coulomb
static const HepDouble coulomb = eplus/e_SI;    // coulomb = 6.24150 e+18 * eplus

//
// Energy [E]
//
static const HepDouble MeV = 1. ;
static const HepDouble  eV = 1.e-6*MeV;
static const HepDouble keV = 1.e-3*MeV;
static const HepDouble GeV = 1.e+3*MeV;
static const HepDouble TeV = 1.e+6*MeV;

static const HepDouble joule = eV/e_SI;         // joule = 6.24150 e+12 * MeV

//
```

```
// Mass [E][T^2][L^-2]
//
static const HepDouble kg = joule*s*s/(m*m);    // kg = 6.24150 e+24 * MeV*ns*ns/(mm*mm)
static const HepDouble  g = 1.e-3*kg;
static const HepDouble mg = 1.e-3*g;

//
// Power [E][T^-1]
//
static const HepDouble watt = joule/s;          // watt = 6.24150 e+3 * MeV/ns

//
// Force [E][L^-1]
//
static const HepDouble newton = joule/m;        // newton = 6.24150 e+9 * MeV/mm

//
// Pressure [E][L^-3]
//
#define pascal hep_pascal                          // a trick to avoid warnings
static const HepDouble hep_pascal = newton/m2;     // pascal = 6.24150 e+3 * MeV/mm3
static const HepDouble bar        = 100000*pascal; // bar    = 6.24150 e+8 * MeV/mm3
static const HepDouble atmosphere = 101325*pascal; // atm    = 6.32420 e+8 * MeV/mm3

//
// Electric current [Q][T^-1]
//
static const HepDouble ampere = coulomb/s;      // ampere = 6.24150 e+9 * eplus/ns

//
// Electric potential [E][Q^-1]
//
static const HepDouble Megavolt = MeV/eplus;
static const HepDouble kilovolt = 1.e-3*Megavolt;
static const HepDouble     volt = 1.e-6*Megavolt;

//
// Electric resistance [E][T][Q^-2]
//
static const HepDouble ohm = volt/ampere;       // ohm = 1.60217e-16*(MeV/eplus)/(eplus/ns)

//
// Electric capacitance [Q^2][E^-1]
//
static const HepDouble farad = coulomb/volt;    // farad = 6.24150e+24 * eplus/Megavolt
static const HepDouble millifarad = 1.e-3*farad;
static const HepDouble microfarad = 1.e-6*farad;
static const HepDouble  nanofarad = 1.e-9*farad;
static const HepDouble  picofarad = 1.e-12*farad;

//
// Magnetic Flux [T][E][Q^-1]
//
static const HepDouble weber = volt*s;          // weber = 1000*Megavolt*ns

//
// Magnetic Field [T][E][Q^-1][L^-2]
```

```
//
static const HepDouble tesla     = volt*s/m2;   // tesla =0.001*Megavolt*ns/mm2

static const HepDouble gauss     = 1.e-4*tesla;
static const HepDouble kilogauss = 1.e-1*tesla;

//
// Inductance [T^2][E][Q^-2]
//
static const HepDouble henry = weber/ampere;    // henry = 1.60217e-7*MeV*(ns/eplus)**2

//
// Temperature
//
static const HepDouble kelvin = 1.;

//
// Amount of substance
//
static const HepDouble mole = 1.;

//
// Activity [T^-1]
//
static const HepDouble becquerel = 1./s ;
static const HepDouble curie = 3.7e+10 * becquerel;

//
// Absorbed dose [L^2][T^-2]
//
static const HepDouble gray = joule/kg ;

//
// Miscellaneous
//
static const HepDouble perCent     = 0.01 ;
static const HepDouble perThousand = 0.001;
static const HepDouble perMillion  = 0.000001;

#endif /* HEP_SYSTEM_OF_UNITS_H */
```