DD

C1

CERN-TH.7175/94

SW 94 10

CLHEP – a Project for Designing a C++ Class Library for High Energy Physics

Leif Lönnblad

Theory Division, CERN
CH-1211 Genève 23, Switzerland
E-mail: lonnblad@surya11.cern.ch

Abstract

CLHEP is project for developing a general C++ class library for High Energy Physics. This paper describes the motivation behind the project together with its current status. Also the future of the project is discussed.

Submitted to Computer Physics Communications

CERN-TH.7175/94
February 1994

1 Introduction

As the number of software projects in High Energy Physics (HEP) using Object Oriented Programming (OOP)¹ in general and C++² in particular increases, the need for a common basic class library becomes more and more obvious. Not only would such a library save a lot of work for software developers by supplying classes corresponding to the most common concepts in HEP; OOP allows for more sophisticated interfaces between different applications compared to conventional procedural programming, and to benefit from this is of course preferable if the different applications are based on a common underlying library.

As an example, consider the interface between the MC++[3] toolkit for event generation and the GISMO[4] detector simulation toolkit, both written in C++. MC++ describes the production and subsequent decay of particles in high energy collisions and GISMO describes the interaction of these particles with a detector.

The aim here is to define particle objects in MC++ which can propagate through the simulated detector in GISMO and either interact with the detector with the help of GISMO or decay “inside” the detector using MC++. The two programs would then be seamlessly integrated, although fully separated in that MC++ needs to know nothing about the detector and GISMO nothing about how particles decay.

This should be compared to the interface between conventional event generator and detector simulation programs like JETSET[5] and GEANT[6], both written in FORTRAN where the interface consists of a large common block where JETSET outputs events as they would look without a detector. GEANT can then read from this common block and simulate the interaction with the detector.

Historically it was also the creation of an interface between MC++ and GISMO which first triggered the idea of writing a common basic class library. At an informal meeting during the “Computers in High Energy Physics” conference in Annecy in 1992 (CHEP92), a number of people attended a discussion and agreed that such a class library was needed, and that a few general classes from the MC++ toolkit should be taken as a starting point for this library.

A year and a half has now passed since this meeting, and “CLHEP – a Class Library for High Energy Physics” has been available in a preliminary “zeroth” version for one year. This paper describes how the development of the library is organized (section 2), and what the current status is (section 3) together with a brief description of the classes in it. Finally section 4 contains a discussion of what should be done with CLHEP in the future.

¹See, eg., reference [1] for an introduction to OOP.

²See, eg., reference [2] for more information about C++ in HEP.

2 Organization of the Project

The organization of the CLHEP project is fairly loose. There is no formal collaboration behind it, it receives no direct funding and there is no one formally in charge of the project. As the classes in CLHEP originated from the MC++ toolkit, one of the authors of MC++ (also the author of this paper) accepted the responsibility of maintaining the code. Using the `hepnet.lang.c++` news group as a forum for discussions, the original code has been improved and new code has been added. The new versions of the library have been made available through the FREEHEP server³.

The reason for this somewhat unconventional (lack of) organization is to allow for as many people as possible to influence the development of the library. Using the `hepnet.lang.c++` news group in principal ensures that everyone (within the HEP community) who is interested can take part in the discussions which will ultimately form the library.

The procedure for adding new code to the library and for making changes in the code already in the library, has been to post an article to `hepnet.lang.c++` with a suggestion. If no one objects (and if the person managing the library has time), the code will be added/changed.

There are no strict coding rules for CLHEP, but the coding style has been influenced by the rules and recommendations found in reference [7]. In addition the header files are supplied with comments in a way which makes them easily converted into UNIX-like “man pages” using the `classdoc`[8] program included in the CLHEP distribution. When installing the library these `classdoc`-processed header files are placed where they are available through the usual UNIX “man” program. Hence typing “man <class name>” gives you a brief (and fairly readable) description of any class in CLHEP. This is presently also the only documentation supplied for CLHEP.

All classes included in CLHEP are accompanied by a small test program which ideally should check that all features of the class are implemented correctly. Before each new release it checked that the library compiles successfully, at least on those platforms available to the person managing the library⁴.

3 Current Status

The current version of CLHEP is 0.13, where the zero indicates that the form and content of the library is still preliminary. It contains around forty classes which can be divided into three groups as follows.

³The FREEHEP server is accessed through anonymous ftp to `freehep.scri.fsu.edu`

⁴See the Appendix for a list of these “supported” platforms.

3.1 Non-HEP classes

The first group contains classes which are not related to HEP, but are useful to have standardized.

- `HepString`:⁵ a simple character string class.
- `HepAList`:⁶ a template-based list class which stores pointers to objects. (A list class which stores copies of objects will be included in a future release).
- `HepConstAList`: a template-based list class which stores pointers to const objects.
- `HepAListIterator` and `HepConstAListIterator`: iterators corresponding to the list classes above. (See figure 4 for an example of how they are used.)
- `HepAListBase` and `HepAListIteratorBase`: abstract base classes for the list and iterator classes above.
- `HepCmdLine`, `HepCmdArg`, `HepCmdArgInt` etc.⁷: classes for handling command line arguments to programs.

Most of these are expected to be made obsolete by corresponding classes in the future ANSI/ISO standard class library for C++.

3.2 HEP related classes

The HEP-related classes are of course the most important ones in CLHEP and are also the ones which are most difficult to design in a way which satisfies everyone. Sometimes there is disagreement on how to interpret the underlying physical concepts, as is the case for *particle* and *track* which are discussed separately below (see section 3.4). In other cases there is disagreement on how these concepts should be implemented, as is the case for three-vectors and Lorentz vectors.

In CLHEP three-vectors are represented by two classes: `Hep3VectorF` and `Hep3VectorD` using float and double components respectively. The corresponding Lorentz vector classes `HepLorentzVectorF` and `HepLorentzVectorD` are then implemented as subclasses, so that `HepLorentzVectorD` “is a” `Hep3VectorD`. In this way (the spatial components of) a Lorentz vector can be used everywhere a three-vector is required.

⁵ All globally visible identifiers in CLHEP are prefixed with `Hep`[7].

⁶ The character string and list classes are based on classes written by Dag Brück.

⁷ Written by Paul Rensing.

```

Hep3VectorD z1(0.0, 0.0, 1.0);           // a unit vector along the z-axis
Hep3VectorD z2 = z1 * 3.14;              // scaling with a real number
z1.rotateY(M_PI/2);                      // rotate around the y-axis
double pt = z2.perp(z1 + z2);            // transverse component wrt. z1 + z2
HepRotation r;                          // matrix for rotation around
r.rotate(M_PI/4, z1 + z2);               // the axis defined by z1 + z2
Hep3VectorF z3 = r * z1;                 // use the rotation matrix
z2 *= r;                                 // z2 *= r <=> z2 = r * z2
z3.transform(r.inverse());               // (inverse) rotation using method
                                           // instead of operator

```

Figure 1: *Example of the syntax for three-vectors in CLHEP.*

The fact that the vectors come in one float and one double version indicates that it might be advantageous to use the *template* mechanism instead, thus easily allowing for `Hep3Vector<anyClass>` as well as `Hep3Vector<float>` and `Hep3Vector<double>`. This is the approach taken by the VMP package [9] which implements general vectors and matrices using templates and gives three- and Lorentz vectors as special cases.

The VMP approach is of course more mathematically stringent but there are some advantages in using the more special-purpose vectors in CLHEP. In addition to the inheritance issues discussed above, the vectors in CLHEP are designed so that the float and double versions can be intermixed in the same way as the corresponding built-in types. This is not handled very well in VMP. Also the fact that templates is a fairly new feature of C++ and the compilers are still not quite stable in their implementation, makes the CLHEP vectors currently more safe.

Although not an official part of CLHEP, the VMP package is distributed *together* with CLHEP. It seems likely that VMP, or something using the same approach, will become a part of CLHEP in the future, either replacing the current vector classes or merging with them.

Simple rotations of three-vectors around the x -, y - or z -axis and pure Lorentz boosts of Lorentz vectors are performed directly through member functions. In addition, general spatial and Lorentz rotations are performed by multiplication with unitary matrices implemented in the `HepRotation` and `HepLorentzRotation`. This is also the case in VMP but here there is currently no guarantee that the rotation matrices really are unitary.

Figures 1 and 2 contain some examples on how the vector and rotation classes are used in CLHEP.

CLHEP contains a couple of classes for defining pseudo random number generators. The strategy taken is to separate the underlying algorithm for generating a flat random number and the algorithms for transforming this flat distribution into more complicated ones. The

```

HepLorentzVectorD p1(1.0, 0.0, 0.0, 1.0);    // a mass-less vector along the x-axis
HepLorentzVectorF p2 = p1 * 3.14;           // scaling with a real number

p1.rotateY(M_PI/2);                          // rotations and other
double pt = p2.perp(p1 + p2);               // manipulations of
HepRotation r;                              // the space like part
r.rotate(M_PI/4, p1 + p2);                  // of the Lorentz vector is
Hep3VectorD z3 = r * p1;                    // inherited from the three-
p2 **= r;                                    // vector classes

HepLorentzVectorD p12 = p1 + p2;             // boost with the spatial component of
p1.boost(-p12/p12.t());                     // p12 scaled by the time-like component
HepLorentzRotation m(-p12/p12.t());          // ie. to the cms. of p1 and p2
m = r * m;                                  // multiplication implicitly converting
p2 **= m;                                   // the HepRotation to a Lorentz rotation
                                           // general Lorentz rotation corresponding
                                           // to "first rotate then boost"

```

Figure 2: *Example of the syntax for Lorentz vectors in CLHEP.*

former should be implemented as a subclass of `HepRandomEngine` where the virtual member function `double flat()` should be implemented to return a number in the interval $]0,1[$ according to a flat distribution. The `HepRandom` class then implements a number of different distributions using a given `HepRandomEngine`⁸. Some of these distributions are given in figure 3. More distributions can, of course, be added by creating subclasses of `HepRandom`.

As of version 0.13 CLHEP also includes classes for doing combinatorial analysis⁹. The basic classes are called `HepLockable` and `HepChooser`. Given lists of objects of a class which inherits from `HepLockable`, `HepChooser` successively generates list of objects which form a unique combination using locks defined in `HepLockable` to ensure the uniqueness.

3.3 Object Oriented Stream I/O

A difficult issue when programming in C++ is how to read and write objects to and from a file or another process. The ANSI/ISO standard will specify standardized so-called I/O-streams, where you can specify the read and write operations on a class by overloading the read (`>>`) and write (`<<`) operators. A major problem with these I/O-streams is that they only handle the basic data members of the objects and in particular the inheritance structure of the classes is completely lost. Another problem is that they do not conserve the pointer structure of objects written out.

⁸By default, an object of the `HepJamesRandom` class implementing the algorithm of reference [10] is used.

⁹Written by Paul Rensing.

```

HepRandom rnd;                                // HepRandom using HepJamesRandom engine
double x = rnd.flat();                        // x in ]0,1[
x = rnd.flat(2.0);                            // x in ]0,2[
x = rnd.flat(-1.0, 1.0);                      // x in ]-1,1[
x = rnd.gauss(0.0, 4.0);                      // Gaussian distribution with mean
                                              // value 0 and variance 4.0
x = rnd.breitWigner(2.0,0.4);                 // Breit-Wigner with mean
                                              // value 0 and variance 0.4
int i = rnd.intFlat(2,5);                     // integer number [2,5[

MyEngine myEngine;                            // use a "better" or
HepRandom myRnd(myEngine);                    // "faster" engine

HepPCharStream input("dumpedRandomState");    // Read a state from a
input >> myRnd;                               // "persistent" stream

```

Figure 3: *Example of the syntax for random number generators in CLHEP.*

CLHEP contains a number of classes defining a strategy for object oriented I/O of objects from/to so-called persistent¹⁰ streams. The aim is to be able to write out large numbers of objects interconnected with pointers from one program and read them back in again in another, preserving the pointer and class structure. A typical HEP example would be to write out an Event object containing a list of pointers to Particle, where Particle is an abstract base class and the pointers refers to objects of derived classes in turn containing lists of pointers to Particle. When read back in again we want to be able to read a "pointer to Particle" and have it pointing to the fully reconstructed object of a derived class.

To achieve this the following classes have been included in CLHEP:

- HepPersistentBase: Abstract base class for all classes to be handled persistently.
- HepPersistentClassDescription: Description of a persistent class: Name, version etc. Is also able to create an object of the corresponding class.
- HepDescriptionList: Global list of all persistent classes in the current process.
- HepPOStream and HepPStream: Abstract base classes for persistent streams.
- HepPCharStream and HepPOCharStream: Derived streams for character based I/O.
- HepPIBinStream, HepPOBinStream: Derived streams for binary I/O.
- HepPIXdrStream and HepPOXdrStream: Derived streams for Xdr I/O.

¹⁰Although not covering all aspects of the concept of *object persistency*.

The `HepPersistentBase` defines a number of virtual methods which must be implemented in a derived class to be handled by the persistent streams. Some of these methods have to do with the run time type information (RTTI) needed; these will be removed with the advent of the ANSI/ISO standard which will contain built-in RTTI. The most important methods defined in `HepPersistentBase` are “void `readPersistently(HepPIStream & is)`” and “void `writePersistently(HepPOStream & os) const`”.

A class that is to be handled by the persistent stream must thus have `HepPersistentBase` as a public base class (directly or indirectly). It must implement the `readPersistently` and `writePersistently` methods so that they first call the corresponding methods of the base class and then read or write each of the members added in the class using the `>>` and `<<` operators. Any member which is either a basic type or (a pointer to) another persistent class can be written out this way.

In addition, each class must specify a function for creating a corresponding object and create an object of the class `HepPersistentClassDescription` which must be added to a global list of classes available in a process. All these things can be implemented using a handful of simple macros placed in the class declaration and in the class implementation file.

Once a class has been defined according to this strategy, implementing the reading and writing for an object of that class becomes more or less trivial, as is shown in figure 4. Of course a lot of things are hidden in the stream classes.

The `HepPOStream` is responsible for output of objects. It has a reference to an ordinary C++ I/O-stream via which it can be connected to a file, a pipe or any other kind of stream. It also contains a list of pointers to objects which it has written and a list of `HepClassDescriptions` which has been written. What actually happens when a pointer to an object is written is the following:

- 1 Check the list of previously written objects.
- 2 If the object was written before, write out an integer corresponding to the order in the list. Done!
- 3 This object hasn't been written before, so put a mark on the stream, signaling a new object. Check the list of previously written class descriptions.
- 4 If this class has been written before, write out an integer corresponding to the order in the list. Go to phase 6.
- 5 This class hasn't been written before so write out the information in the class description (name, version etc.) check the list of previously written class description for the base class of this class, go to phase 4
- 6 Ask the object to write itself on the stream. Put a mark on the stream to signal the end of a new object.

```

void write(MyPersistentClass * array[3]) {
// Write out an array of four pointers to persistent objects

    HepPOCharStream output("SomeFile");           // Create a persistent stream
                                                    // writing to a file

    output << 4 << array[0] << array[1]           // write out the number of objects
        << array[2] << array[3];                 // and the pointers to them
}

void read(istream & is) {
// read a number of objects from an ordinary C++ io-stream
// do something sensible with them

    HepPICharStream input(is);                    // Create a persistent stream reading
                                                    // from the standard istream

    int n;
    input >> n;                                    // read the number of objects present

    HepAList<MyPersistentClass> list;              // create a list
    MyPersistentClass * p;                        // and a pointer for later use

    for ( int i = 0; i < n; i++ ) {                // loop over number of objects
        input >> p;                                // read a pointer to the reconstructed
        list.append(p);                            // object and add it to the list
    }

    HepAIterator<MyPersistentClass> it(list);      // Here is an example on how to use
                                                    // the iterators in CLHEP
    while ( p = it() ) {                          // loop over all objects in the list
        p->somethingSensible();                    // and do something sensible with each
    }

                                                    // The objects were allocated with
    HepAListDeleteAll(list);                      // 'new' in the stream so we have
                                                    // to delete them afterwards
}

```

Figure 4: *Example of the syntax for “persistent” I/O in CLHEP.*

When a HepPIStream is then told to read a pointer or a reference to an object, the following will happen:

- 1 Check the stream for a “new object marker”
- 2 If no marker is found, the object has been read before. Read an integer from the stream, and get the corresponding object in the list of previously read objects. If we are reading a reference to an object - throw an exception, otherwise if the argument is a pointer to a public base class of the object, return a pointer to the object. Done!
- 3 Read an integer from the stream. If this corresponds to a HepClassDescription in the list of previously read class descriptions, go to phase 5.
- 4 This class has not been read before. Read in a class description and go to phase 3 to get the description of the base class.
- 5 If the class description of the object to be read in is not found in the global list of available classes (HepDescriptionList), get the class description of closest base class present.
- 6 If we are reading a reference to an object and this object is at least a public base class of the object to be read from the stream, ask it to read itself from the stream. If we are reading a pointer to a public base class of the object to be read, ask the class description to create a corresponding object, ask this object to read itself from the stream and return the pointer. Otherwise throw an exception.
- 7 Parse the stream for a “end of object” marker. If any “new object” markers are found, read this object, recursively starting from phase 3, and add it to a list of orphans (for subsequent deletion if no pointers to this object has been read when the stream is destroyed). Done.

The reason for the last phase in the read procedure is to be able to read the parts of an object corresponding to base class even if the full class of the object is completely unknown to (ie. not linked with) the current process. To avoid errors, if the full object has written out pointers to other objects which are referenced by later read operations, all these objects must be taken care of by the stream. This behavior is optional and by default the HepPIStream is in a “pedantic” state where only objects of known classes are allowed.

3.4 Particles and Tracks

As mentioned above, the introduction of a basic particle class in CLHEP has caused some discussion. One of the reasons for disagreement is that people make different associations to the word. An experimentalist would think of a track reconstructed in a detector and a theoretician would think of the actual particle which left the track. It is clear that a track and a particle are different concepts and it would therefore be obvious that they should be represented by two different classes in C++.

One could, however, imagine someone writing a program for analysis of experimental data consisting of events with reconstructed tracks where he only uses things like the energy or charge of the tracks. Such an analysis program can, of course, be equally well applied to events containing “theoretical” particles produced by a event generator program. So even if tracks and particles are conceptually different, it *maybe* still would make sense to have them derived from a common base class in C++.

CLHEP contains a suggestion for such a common abstract base class - `HepBaseParticle` - which is designed according to the principle of minimum number of data members and maximum number of virtual member functions. In this way a particle or a track class can be derived without any constraints on how the data is stored. And since the virtual member functions have sensible default return values in `HepBaseParticle`, only those which correspond to information available in the derived class need to be re-implemented. If, in addition, all derived classes have a constructor taking a reference to `HepBaseParticle` as argument, it is simple to convert one kind of particle to another.

CLHEP also includes a `HepInterfaceParticle` to be used as an external representation of a `HepBaseParticle`. It has the same member functions as the latter, but in addition it also has corresponding data members. This means that the analysis program in the example above does not have to know anything about the particle classes used in the event generator program as long as they are converted into `HepInterfaceParticles` before being written to the persistent stream.

As an example of a different approach, CLHEP contains an extra sub-directory with classes written by Paul Rensing for the BFAST group[11]. Here a particle base class is defined which is more specialized for analysis.

4 The Future of CLHEP

Although CLHEP has evolved since the first release more than a year ago, the basic structure has not really changed and it is still considered preliminary and called “version zero”.

The reason for the slow progress is probably the current (lack of) organization. Not very many people have participated in the discussions in the `hepnet.lang.c++` news group and even fewer have contributed with any code.

If CLHEP is going to evolve into a standard class library for HEP, more people will have to join the discussion and try to influence the design. And, above all, more people must become involved with the actual code design. To achieve this it may be necessary to find a better forum for discussion. Also the situation with only one person managing the code will have to change to allow for more flexibility when including new code.

It may even be necessary to create a more formal project, possible with some kind of “official” sanctioning from one or several of the major HEP laboratories or experiments.

As it stands, CLHEP does not claim to be *the* class library for High Energy Physics, but it could serve as a good starting point for such a library. Anyone involved with writing software in HEP is more than welcome to have a look at it¹¹ and use it, and is encouraged to send their comments to the `hepnet.lang.c++` news group.

Acknowledgements

I would like to thank all people participating in the discussions about CLHEP in the `hepnet.lang.c++` news group and especially Dag Brück, Anders Nilsson, Paul Rensing, Ernest Prabhakar and Irwin Sheer who have contributed with code to the library.

References

- [1] P.F. Kunz, *Particle World* **2** (1991) 122.
- [2] A large amount of information about C++ and its use in HEP, is available on the “World Wide Web” using the ULR `http://info.desy.de/user/projects/C++.html`
- [3] L. Lönnblad and A. Nilsson, *Comp. Phys. Comm.* **71** (1992) 1.
- [4] W. Atwood et al., “The Gismo Project”, *C++ Report* **5** (Mars-April 1993) 38, SLAC-PUB-6135.
- [5] T. Sjöstrand, JETSET 7.3 program and manual, CERN-TH.6488/92;
T. Sjöstrand, *Comp. Phys. Comm.* **39** (1986) 347;
T. Sjöstrand and M. Bengtsson, *Comp. Phys. Comm.* **43** (1987) 367.
- [6] R. Brun et al., “GEANT: Simulation Program for Particle Physics Experiments, User Guide and Reference Manual”, CERN-DD/78/2, July 1978.
- [7] E. Nyquist and M. Henricson, “Programming in C++. Rules and Recommendations”, Ellemtel document no. M900118, available through anonymous ftp to eg. `ftp.desy.de` in the file `pub/c++/misc/c++.rules.ps`
- [8] D. Brück, classdoc program and manual, available through anonymous ftp to `control.lth.se` in the file `pub/c++/classdoc.shar`

¹¹The Appendix describes how to obtain and install CLHEP.

- [9] E.N. Prabhakar and J.H. Panetta, "VMP - a Vector Manipulation Package", program and documentation available through anonymous ftp to `cithe501.cithec.caltech.edu` in the file `pub/sushi/vmp-2.0.tar.Z`
- [10] F. James, *Comp. Phys. Comm.* **60** (1990) 329.
- [11] D. Aston et al., "BFAST Computing Recommendations", SLAC-BABAR-NOTE-118, Nov 1993.
- [12] GNU source code is available via anonymous ftp to `prep.ai.mit.edu` in the directory `pub/gnu`. It is also available on CDROM from the Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139.

All code in the CLHEP library is written by Leif Lönnblad and Anders Nilsson except where stated otherwise.

Appendix: Technical Information

The CLHEP is obtainable through anonymous ftp to `freehep.scri.fsu.edu` in the file `hep-projects/CLHEP/CLHEP-0.13.tar.Z`. After getting this file via ftp you should type “uncompress CLHEP-0.13.tar.Z” and “tar xf CLHEP-0.13.tar” which will give you a directory called CLHEP-0.13 containing the CLHEP distribution.

Among other things, this directory contains a script called `configure` which, when run, will try to figure out how your system is configured and will create a suitable `Makefile`. Typing “make check” after that will compile the library, and compile and run a number of test programs, which, if the library is correctly compiled, should run quietly without any output.

If nothing has gone wrong, the library can be installed by typing “make install_all”. This will install all the header files in `/usr/local/include/CLHEP`¹² and the archived object files in `/usr/local/lib/libCLHEP.a`. In addition the `classdoc` program will be installed in `/usr/local/bin` and the `classdoc`-processed header files as UNIX man-pages in `/usr/local/man/man3`.

The directory also contains a `README` file containing more technical information about the library and how to install it.

The code should in principle run on any platform with the GNU[12] C++ compiler of version 2.4.5 or later. In practice, however, the code has only been tested on the following platforms:

- SparcStation (under SunOS) with gcc-2.5.8 and libg++-2.5.3
- HP 9000/730 with gcc-2.4.5 and libg++-2.4
- Silicon Graphics iris4d with gcc-2.4.5 and libg++-2.4
- DECstation 3100 with gcc-2.5.8 and libg++-2.5.3
- DEC Alpha with the DEC C++ compiler

¹²If another installation directory is desired, the `/usr/local` prefix can be substituted by, eg., “make install_all prefix=/some/other/directory”