Getting Started with DANA

D. Lawrence
Jefferson Lab

May 23, 2005

Abstract

The Hall-D **D**ata **ANA**lysis framework (DANA) provides the mechanism by which various pieces of the reconstruction software are brought together to fully reconstruct the data. This document is intended for user's new to the DANA system. This includes both end user's who simply want to get at the data for their own analysis, and person's writing reconstruction code for a particular subsystem. It is purposely kept short to minimize your TTR (Time To Results).

Contents

1 Introduction **3**

2 Getting and Compiling the Source Code **3**

3 Running the software **4**
3.1 Generating a Data File 4
3.2 *hd_dump* 6
3.3 *hd_ana* 6
3.4 *hdview* 6

4 Adding your own code **6**
4.1 "I just want to make a histogram!" 7
4.2 Writing a Factory 7

5 Questions **11**

A Source for *hd_ana* program **12**

B Glossary of Terms **17**

C Other Useful Documents **18**

1 Introduction

The DANA system is centered around the idea of data factories. In the DANA system, a factory is an object which takes one or more inputs and produces a single output. The inputs and outputs are lists of objects. So, for example, if I wanted to produce a list of showers in the forward lead glass calorimeter (FCAL), I would first need a list of hits in the FCAL. Similarly, in order to produce a list of tracks, I would first need a list of hits in the CDC, FDC, etc. In general, factories get their inputs from other factories. End point users (persons doing an analysis and not trying to produce a list for anyone else) get at the data in the same way, by accessing the factories.

The factory objects themselves provide "data on demand" [1]. What this means is that when an event is read in, nothing is analyzed until the data is actually requested from the factory. If the program being run is only interested in Barrel Calorimeter (BCAL) data, then there is no need to do things like full scale tracking or FCAL reconstruction on the event. Since factories get their input from other factories, a linked "web" of factories is naturally formed with each knowing what factories it needs to access for its inputs.

2 Getting and Compiling the Source Code

The Hall-D reconstruction code is kept in a CVS repository on the Hall-D group disk (/group/halld/Repositories/cvsroot) at Jefferson Lab. To access it, you need an account on the JLab CUE which belongs to the halld unix group[2]. Follow these steps to checkout and compile the code:

- **Create working directory:** All of the source code and binaries will reside in this directory. It can be named anything and placed anywhere. In my account, I use */home/davidl/HallD*.
- **Set HALLD_HOME environment variable:** The HALLD_HOME variable should be set to the working directory you just made. The makefile system [3] uses this to find the source code and place the resulting binaries. So, for my account this is set to */home/davidl/HallD*.

  ```
  setenv HALLD_HOME /home/davidl/HallD
  ```

  (I use tcsh in these instructions I'll leave it to *bash* users to translate where appropriate.)
- **Checkout the source:** Go into your working directory and checkout the code by doing the following:

  ```
  cd $HALLD_HOME
  setenv CVS_RSH `which ssh`
  setenv CVSROOT cvs.jlab.org:/group/halld/Repositories/cvsroot
  cvs co src include
  ```

[1] This concept is also called "just in time".
[2] contact the JLab Computer Center if you need don't already have such an account
[3] See GlueX-doc-473 on the BMS system

Note that this assumes the account you're issuing the command from has the same username as your JLab CUE account. If not, prefix *CVSROOT* with your JLab account name followed by an '@' [4].

- **Move the** *include* **directory:** Hopefully this step will be removed in the near future. But for now, you'll need to move the include directory you just checked out into the src directory (that you also just checked out).

```
mv include src
```

- **Compile the libraries:** Go into the *src/libraries* directory and run gmake:

```
cd src/libraries
gmake
```

This should build all of the libraries and place them in *$HALLD_HOME/lib/$OSNAME* where $*OSNAME* is the uname of the system you're working on (e.g *Linux*).

- **Compile the executables:** Go into the *src/programs/Analysis* and run *gmake* in the *hd_dump*, *hd_ana*, and *hdview* directories:

```
cd $HALLD_HOME/src/programs/Analysis
gmake -C hd_dump
gmake -C hd_ana
gmake -C hdview
```

The executables will be placed in the *$HALLD_HOME/bin/$OSNAME* directory. These programs do the following:

  - *hd_dump* Dump ASCII output to screen from all factories
  - *hd_ana* Generates a couple of example *ROOT* histograms
  - *hdview* Graphical drawing of data

3 Running the software

All of these programs use the same command line parser routine that is part of DANA. This just means, is that any command line argument that does not begin with a "-" is assumed to be a data file[5]. Each of the programs will loop over all events in all of the data files specified on the command line. See the sections below for examples.

3.1 Generating a Data File

If you already have an HDGeant produced data file then you can probably skip this section. Note though that HDDM can be very picky about version numbers in files so if you have an old file, the executables may simply exit with an error.

Here, I'll assume you have a working *hdgeant* executable. For the sake of simplicity, I'll show how to use the single particle generator built into *hdgeant*. The only thing you really

[4] e.g. joe@cvs.jlab.org:/group/halld/Repositories/cvsroot

[5] At this point in time, the only format we have for data files is HDDM

need is a file called *control.in* in the directory you run hdgeant from. Here's an example one that throws 1.5GeV μ^+s from the center of the target (z=65cm) at randomly chosen angles. See the *GEANT* documentation for descriptions of the uncommented "cards".

Listing 1: control.in

```
TIME 1. 1. 1000000
TRIG 1000000
RUNG 9999

C The following card enables single-track generation (for testing)
C Note: if you REALLY want the specified theta/phi for every event,
C you must add 100 to the particle type. Otherwise, theta and phi
C will be randomly chosen.
C    particle  energy  theta  phi  vertex(x y z)
KINE    5      1.5     10.  265.    0. 0. 65.

RNDM 172

CUTS 1e-4 1e-4

SWIT 0 0 0 0 0 0 0 0 0 0

C The following card enables the GelHad package (from BaBar)
C   on/off  ecut   scale   mode   thresh
GELH  1     0.2     1.0     4     0.160

HADR 1
CKOV 1
LABS 1
LOSS 1
MULS 1
MUNU 1
PAIR 1
PFIS 1
PHOT 1
RAYL 1
STRA 1
SYNC 1

C The NOSECONDARIES card is specific to HDGeant. Setting it to a non-zero
C value will skip calls to GSKING()
NOSECONDARIES 1

C The BFIELD card is specific to HDGeant. Setting it to some value
C (in Tesla) will cause the tracking to use a completely homogeneous
C magnetic field with this value in the Z (beam) direction. Otherwise,
C the TOSCA generated field map(dsolenoid.table) will be used.
BFIELD -2.0

END
```

Run *hdgeant* without any arguments. It will produce several files, but the only one

you're interested in is called *hdgeant.hddm*.

3.2 *hd_dump*

The *hd_dump* program is a very basic one that can be used to dump single events in ASCII format to the terminal. Run it with no arguments to see a usage statement. Basically, to see the output of any factory, just use the $-D$ option. For example, the following will dump the contents of both the $DBCALHit$ and $DFCALHit$ factories:

```
hd_dump -DDBCALHit -DDFCALHit hdgeant.hddm
```

This program is useful just to see that you're able to read the data. It relies on the factory objects themselves to format the output. Since factories are not required to implement a "toString()" method, some of them may not produce very useful output from *hd_dump*. At the very least, the factory name and the number of elements will always be printed.

3.3 *hd_ana*

The *hd_ana* program is intended to be the definitive example program for end user analysis. It opens a ROOT file, creates a few histograms, and then fills them using data obtained through $DANA$. If you're writing a new analysis program, you probably just want to copy the source from this directory and edit from there.

It is worth compiling and running to test that things are working. Run it like this:

```
hd_ana hdgeant.hddm
```

3.4 *hdview*

The *hdview* program provides a simple 2-dimensional graphical representation of events. It was developed to help visualize things for tracking and so may be of limited use for other systems. Nevertheless, it has some basic diagnostic value. Run it just like all of the other programs:

```
hdview hdgeant.hddm
```

4 Adding your own code

At this point, you should have a working copy of the Hall-D source code and an *hdgeant* produced data file (*hdgeant.hddm*) to work with. Modifying the source can be broken up into two groups:

- **End User:** Basically, these are executable programs. If you're just looking to make some histograms or *trees*, then you belong in this group. Projects in this group create a class derived from $DEventProcessor$. See section 4.1.

- **Factories:** Projects in this group perform some piece of reconstruction or analysis that is made available to other factories or end user's. These are kept in libraries, usually with other related factories. See section 4.2.

4.1 "I just want to make a histogram!"

Oh you do, do you? Good, you've come to the right place. The best tutorial for this is actually to look at the example program *hd_ana*. The source for this program is kept in *$HALLD_HOME/src/programs/Analysis/hd_ana*. For convienience, the source has been included in this note in appendix A. The basic idea is this: You need to create a class which inherits from the $DEventProcessor$ class. You can then overide any of the methods corresponding to the 5 phases of the data processing:

- **init(void)** Called once at initialization
- **brun(int runnumber)** Called anytime the run number changes. Guaranteed to be called once before the first call to $evnt()$.
- **evnt(int eventnumber)** Called every event.
- **erun(void)** Called when the run number changes (iff $brun()$ was already called) or after all events have been processed.
- **fini(void)** Called once after all events have been processed.

You create a single instance of your derived class and pass a pointer to it to the framework. Control is then handed over to the framework which will call the above methods of your class at the appropriate times.

Inside your $evnt()$ method, you will request data objects from the framework. The framework will look through its list of factories until it finds the one which produces the requested data type and returns a list of those objects to you.

4.2 Writing a Factory

If you are writing reconstruction code that produces output for others to use, then you should write a factory. A factory consists, at a minimum, of 3 files:

- the produced object header
- the factory object header
- the factory object methods

These can be produced using the $mkfactory$ script which resides in the *scripts* directory in CVS [6]. Invoke the script with the name of the data object you'd like to generate with the factory. For example:

```
mkfactory DParticle
```

[6] cvs -d cvs.jlab.org:/group/halld/Repositories/cvsroot co scripts/mkfactory

would produce 3 files named *DParticle.h*, *DFactory_DParticle.h*, and *DFactory_DParticle.cc*.

The 2 header files should be copied to the $*HALLD_HOME/src/libraries/include* directory. The source should be placed in the appropriate subsystem directory. For example, a factory dealing with BCAL should be placed in the $*HALLD_HOME/src/libraries/BCAL* directory. In that same directory, there is a file named something like *BCAL_init.cc* which looks like this:

Listing 2: BCAL_init.cc

```
// $Id: BCAL_init.cc,v 1.2 2005/04/06 19:13:48 davidl Exp $

#include "DEvent.h"
#include "DFactory_DBCALHit.h"

derror_t BCAL_init(DEvent *event)
{
   /// Create and register BCAL data factories
   event->AddFactory(new DFactory_DBCALHit());

   return NOERROR;
}
```

BCAL_init() (as well as all of the other subsystems' $*$*_init* routines) gets called when the *DEventLoop* object is instantiated in the *main*() routine of all programs using DANA. This is how your factory gets added to the list of factories kept by the DANA.

One thing to keep in mind is that a program may be multi-threaded. In this case, multiple objects of your factory class may be created. Basically what this means is that you should just avoid using global or static variables.

The place where you actually want to place your code for the factories is in the *init*(), *brun*(), *evnt*(), *erun*(), and *fini*() methods[7] of your new DFactory class. The most common usage, however, will be to read calibration constants in *brun*() and then do the actual reconstruction in *evnt*().

The factory will produce data in the form of objects. More specifically, the factory will fill an STL *vector* of pointers to objects which the factory creates. The DANA framework will automatically delete these objects when the next event is read in. So, the first thing you need to do is finish the definition of the class by adding the data fields. Listing 3 shows an example of a class *DMCReconstructed* produced by a factory. The *mkfactory* script created everything in this file except the lines after *HDCLASSDEF*(*DMCReconstructed*). The *HDCLASSDEF*(*DMCReconstructed*) is a DANA defined macro that places a couple of methods in the class which it needs. It must be placed in the public area of the class.

Notice that each data element in listing 3 has a comment prefaced with "/// <". This special comment delimiter is used by the Doxygen documentation system to present the

[7] see section 4.1

class definition in the web-based documentation in the proper place.

Listing 3: DMCReconstructed.h

```
// $Id: DMCReconstructed.h,v 1.3 2005/04/25 19:20:39 davidl Exp $
//
//    File: DMCReconstructed.h
// Created: Sun Apr  3 12:28:45 EDT 2005
// Creator: davidl (on Darwin Harriet.local 7.8.0 powerpc)
//

#ifndef _DMCReconstructed_
#define _DMCReconstructed_

#include "DFactory.h"

class DMCThrown;

class DMCReconstructed{
   public:
      HDCLASSDEF(DMCReconstructed);

      int type;          ///< GEANT particle ID
      float q;           ///< electric charge
      float p;           ///< Total momentum in GeV/c
      float E;           ///< Total energy in GeV
      float theta,phi;   ///< Inital theta and phi angles in radians
      float x,y,z;       ///< Vertex position in cm
      float mass;        ///< Mass in GeV/c^2
      int thrownid;      ///< index to closest match in DMCThrown
      float thrown_delta_p;   ///< Magnitude of momentum diff. with thrownid

      void FindClosestThrown(vector<const DMCThrown*> &mcthrowns);
};

#endif // _DMCReconstructed_
```

The next step is to create objects in your factory's *evnt*() method and add them to the *_data* vector of your factory. The variable *_data* is a vector whose type is based on the class of the objects the factory creates. Listing 4 shows the *evnt*() method for the factory that creates the *DMCReconstructed* objects. This factory accesses data from two other factories in order to produce its output. The key thing to notice is that near the bottom of the routine is a line *_data.push_back*(*mcreconstructed*). This is how the objects are stored so that DANA can access them. DANA will take care of casting the pointers as *const* before delivering them to others. DANA will also take care of deleting the objects later, when the next event is read in.

Listing 4: DFactory_DMCReconstructed.cc

```
// $Id: DFactory_DMCReconstructed.cc,v 1.3 2005/05/06 13:59:08 davidl Exp $
//
```

```
//     File: DFactory_DMCReconstructed.cc
// Created: Sun Apr  3 12:28:45 EDT 2005
// Creator: davidl (on Darwin Harriet.local 7.8.0 powerpc)
//

#include "DMCTrackCandidate.h"
#include "DMCThrown.h"
#include "DFactory_DMCReconstructed.h"
#include "DEvent.h"

//------------------
// evnt
//------------------
derror_t DFactory_DMCReconstructed::evnt(int eventnumber)
{
   // For now, we just copy from the MCTrackCandidates. Eventually,
   // a track fitter will be implemented.
   vector<const DMCTrackCandidate*> mctc;
   event->Get(mctc);

   vector<const DMCThrown*> mcthrowns;
   event->Get(mcthrowns);

   for(unsigned int i=0; i<mctc.size(); i++){
      const DMCTrackCandidate *mctrackcandidate = mctc[i];
      DMCReconstructed *mcreconstructed = new DMCReconstructed;

      mcreconstructed->type = 0;
      mcreconstructed->q = mctrackcandidate->q;
      mcreconstructed->p = mctrackcandidate->p;
      mcreconstructed->E = 0.0;
      mcreconstructed->theta = mctrackcandidate->theta;
      mcreconstructed->phi = mctrackcandidate->phi;
      mcreconstructed->x = 0.0;
      mcreconstructed->y = 0.0;
      mcreconstructed->z = mctrackcandidate->z_vertex;
      mcreconstructed->mass = 0.0;
      mcreconstructed->FindClosestThrown(mcthrowns);

      _data.push_back(mcreconstructed);
   }

   return NOERROR;
}

//------------------
// toString
//------------------
const string DFactory_DMCReconstructed::toString(void)
{
   // Ensure our Get method has been called so _data is up to date
   Get();
```

```
    if(_data.size()<=0)return string(); // don't print anything if we have no data!

    printheader("row: type:  q:    p:    E: theta:   phi:   mass:    x:
y:  |  z:");

    for(unsigned int i=0; i<_data.size(); i++){

       DMCReconstructed *mcreconstructed = _data[i];

       printnewrow();

       printcol("%d", i);
       printcol("%d", mcreconstructed->type);
       printcol("%+d", (int)mcreconstructed->q);
       printcol("%3.1f", mcreconstructed->p);
       printcol("%3.1f", mcreconstructed->E);
       printcol("%1.3f", mcreconstructed->theta);
       printcol("%1.3f", mcreconstructed->phi);
       printcol("%1.3f", mcreconstructed->mass);
       printcol("%2.2f", mcreconstructed->x);
       printcol("%2.2f", mcreconstructed->y);
       printcol("%2.2f", mcreconstructed->z);

       printrow();
    }

    return _table;
}
```

The bottom half of listing 4 is the factory's *toString*() method. This is used by *hd_dump* to print the factory contents to the screen. A few routines are provided to the factory by the *DFactory_base* base class to help make this easier. These are *printheader*, *printnewrow*, *printcol*, and *printrow*. The example illustrates how these are used.

5 Questions

For general discussion or questions which others may benefit from, you are encouraged to post to the GlueX forums at:

http://tantalus.phys.uregina.ca/gluex/modules.php?name=Forums

Please refer all other questions to David Lawrence at *davidl@jlab.org*.

A Source for *hd_ana* program

Listing 5 shows the *main()* routine for the hd_ana program. It only has three lines. It just creates a MyProcessor object and a DEventLoop object and then it tells the DEventLoop object to run over all events, calling the methods of the MyProcessor object when needed. Notice that the command line arguments are given to the DEventLoop object when it is created. It will parse them for ,among other things, the file names to read events from.

Listing 5: hd_ana.cc

```
// Author: David Lawrence  June 25, 2004
//
//
// hd_ana.cc
//

#include "MyProcessor.h"
#include "DEventLoop.h"

//-----------
// main
//-----------
int main(int narg, char *argv[])
{
   // Instantiate our event processor
   MyProcessor myproc;

   // Instantiate an event loop object
   DEventLoop eventloop(narg, argv);

   // Run though all events, calling our event processor's methods
   eventloop.Run(&myproc);

   return 0;
}
```

Listing 6 shows the header file which defines the *MyProcessor* class for this example. It defines 5 routines corresponding to the 5 phases of event processing described in section 4.1. Here, the *brun*() and *erun*() routines are defined right in the header to do nothing and just return no error[8]. In fact, those lines could be left out completely and you'd get the same effect since the *DEventProcessor* base class defines them to do the same thing. They are defined in this example just to remind you of the format in case you actually need to use them.

Pointers for the ROOT file and a few histograms are defined here as well. These pointers don't have to be kept here, that's just a style choice.

Listing 6: MyProcessor.h

```
// Author: David Lawrence  June 25, 2004
//
//
// MyProcessor.h
//
/// Example program for a Hall-D analyzer which uses DANA
///

#include "DEventProcessor.h"
#include "DEventLoop.h"

#include <TFile.h>
#include <TH1.h>
#include <TH2.h>


class MyProcessor:public DEventProcessor
{
   public:
      derror_t init(void);                          ///< Called once at program start.
      derror_t brun(int runnumber){return NOERROR;}  ///< Called everytime a new run nu
      derror_t evnt(int eventnumber);               ///< Called every event.
      derror_t erun(void){return NOERROR;}          ///< Called everytime run number c
      derror_t fini(void);                          ///< Called after last event of la

      TFile *ROOTfile;
      TH2 *cdc_y_vs_x;
      TH1 *fcal_y_vs_x, *fcalhitE;
};
```

[8] *NOERROR* is defined in *derror.h* which is included from *DEventProcessor.h*.

Listing 7 is the file that contains the real meat of the program. The *init*(), *evnt*(), and *fini*() methods are defined. I'll describe each of them a little here:

derror_t MyProcessor::init(void) This routine is where the ROOT file is opened and the histograms are defined. The first line: *eventLoop− >PrintFactories();* just dumps a list of all of the registered factories to the screen. It isn't required. The rest of the lines are just ROOT. See the ROOT documentation for details.

derror_t MyProcessor::evnt(int eventnumber) This routine gets called every event and is where the histograms are filled. In this example, two types of data are requested from the *DEventLoop* object, *eventLoop*[9]. First, the variables *cdchits* and *fcalhits* are declared[10]. Next, the eventLoop is asked to get these data types from whatever factory produces them.

In the next section, the histograms are filled inside of two *for*() loops which loop over the elements returned by the *eventLoop− > Get*() calls.

The last line, *eventLoop− > PrintRate*(), updates the event processing rate on the screen. It only prints to the screen about once per second and so is safe to call every event. It is not required though.

derror_t MyProcessor::fini(void) This routine just flushes the histograms to the ROOT file and then closes it.

Listing 7: MyProcessor.cc

```
// Author: David Lawrence  June 25, 2004
//
//
// MyProcessor.cc
//

#include <iostream>
using namespace std;

#include "MyProcessor.h"
#include "hddm_s.h"

#include "DCDCHit.h"
#include "DFCALHit.h"

//-----------------------------------------------------------------
// init   -Open output file here (e.g. a ROOT file)
//-----------------------------------------------------------------
derror_t MyProcessor::init(void)
{
   // Print list of factories
```

[9]The *eventLoop* variable is part of the *DEventProcessor* class and so is inherited by the *MyProcessor* class. It is set by DANA before calling any of the *MyProcessor* methods.

[10]Note that since we are asking for the DCDCHit and DFCALHit objects, their respective header files are included at the top of the file.

```
   eventLoop->PrintFactories();

   // open ROOT file
   ROOTfile = new TFile("hd_ana.root","RECREATE","Produced by hd_ana");
   cout<<"Opened ROOT file \"hd_ana.root\""<<endl;

   // Create histogram
   cdc_y_vs_x  = new TH2F("cdc_y_vs_x","CDC Y vs. X",200, -70.0, 70.0, 200, -70.0, 70.0)
   fcal_y_vs_x = new TH2F("fcal_y_vs_x","FCAL Y vs. X",200, -100.0, 100.0, 200, -100.0,
   fcalhitE = new TH1F("fcalhitE","fcal single detector energy(GeV)",100, 0.0, 6.0);

   return NOERROR;
}

//------------------------------------------------------------------
// evnt   -Fill histograms here
//------------------------------------------------------------------
derror_t MyProcessor::evnt(int eventnumber)
{
   vector<const DCDCHit*> cdchits;
   vector<const DFCALHit*> fcalhits;
   eventLoop->Get(cdchits);
   eventLoop->Get(fcalhits);

   for(unsigned int i=0;i<cdchits.size();i++){
      const DCDCHit *cdchit = cdchits[i];
      float x = cdchit->radius*cos(cdchit->phim);
      float y = cdchit->radius*sin(cdchit->phim);
      cdc_y_vs_x->Fill(y,x);
   }

   for(unsigned int i=0;i<fcalhits.size();i++){
      const DFCALHit *fcalhit = fcalhits[i];
      fcal_y_vs_x->Fill(fcalhit->y,fcalhit->x);
      fcalhitE->Fill(fcalhit->E);
   }

   eventLoop->PrintRate();

   return NOERROR;
}

//------------------------------------------------------------------
// fini   -Close output file here
//------------------------------------------------------------------
derror_t MyProcessor::fini(void)
{
   ROOTfile->Write();
   delete ROOTfile;
   cout<<endl<<"Closed ROOT file"<<endl;

   return NOERROR;
```

```
}
```

B Glossary of Terms

attribute: In terms of a C++ object, an attribute is a data member of a class. See also *method*.

casting, typecasting: A way of instructing the C/C++ compiler to treat a variable as though it is of a certain type of data. This is often used to cast pointers so that the data being pointed to is treated as a certain type.

class: A *class* is a definition of an object. An object is just the implementation of a class. Hence, one writes a class, but when the program runs, there may be many instances(i.e. objects) of that class. A class consists of two types of members: attributes(data) and methods(algorithms).

inheritance: Once a class is defined, another class can be defined which extends or *inherits* from it. The class which inherits is called a *subclass* of the *base class* it inherited from. The idea is that the subclass contains everything that the base class does, but can then be specialized or enhanced. This is a powerful feature that allows for *polymorphism*.

method: An algorithm belonging to a class. To a FORTRAN or C programmer, this would simply be a subroutine or a function. Inside the method, the data memebers of the object are automatically available as local variables.

OO, Object-Oriented: A software design paradigm in which data and algorithms(methods) are combined into single entities – objects. The objects communicate to each other through well defined interfaces which allow objects to keep some data private and inaccessible to other objects. FORTRAN and C are condsidered *procedural* languages while C++ is *object-oriented*.

pointer: An address in RAM memory. In languages such as FORTRAN which do not use dynamic memory allocation, pointers are not accessible to the programmer. In C/C++, the program can decide while it is running that it needs a new block of memory and request it from the system. The system will return a pointer to the new memory block (or the new object for C++).

polymorphism: This is a property that allows a subclass to be used as though it were the base class. For instance, if a class *shape* exists, one can define two subclasses called *circle* and *square*. Both circle and square objects can be treated as objects of type *shape* (e.g kept in a list of *shape* objects.

STL, Standard Template Library: Some standard C++ templates used mainly for dealing with arrays of objects. Note that templates are an integrated part of C++ and are available even without STL.

subclass: See *inheritance*.

vector: A randomly accessible array of objects. This is defined as part of the STL. By contrast, the STL also defines a *list* which is a linked list of objects.

C Other Useful Documents

- GlueX-doc-64 Geometry Specifications for Hall D (HDDS)
- GlueX-doc-65 HDDM — Hall D Data Model