# Hall A Analyzer Introduction & Tutorial

Ole Hansen

Jefferson Lab

Hall A & C Analysis Workshop
June 26, 2017

# Before We Get Started

Please update the "tutorial" subdirectory in your virtual machine

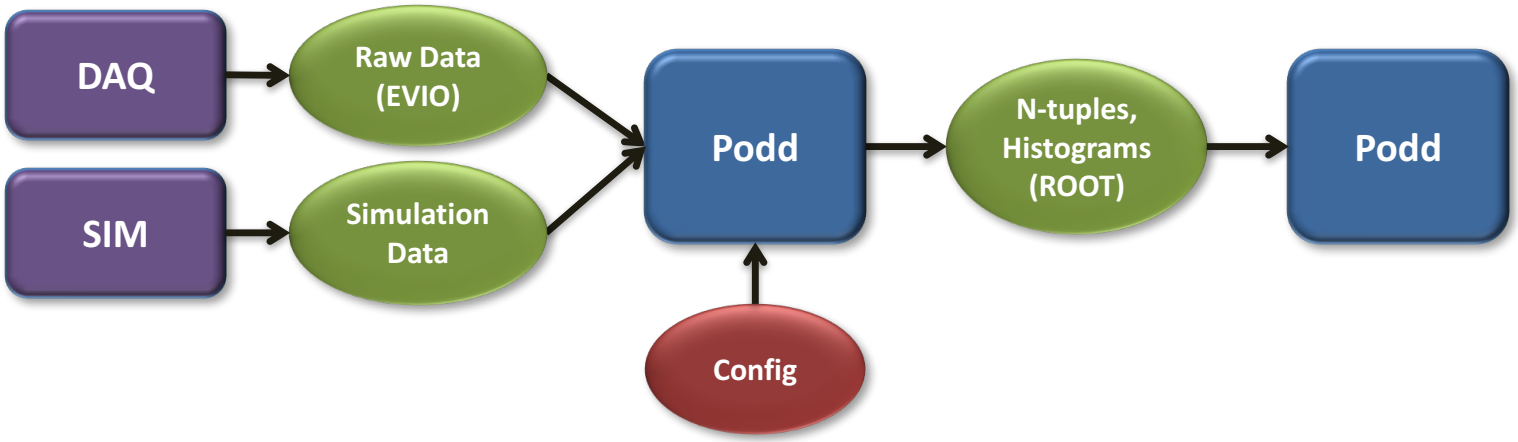
```
[me@centos7 ~]$ cd ~/tutorial
[me@centos7 tutorial]$ git pull
```

Also, this is a good time to personalize your git configuration

```
[me@centos7 ~]$ git config --global user.name "Your Name"
[me@centos7 ~]$ git config --global user.email you@jlab.org
```

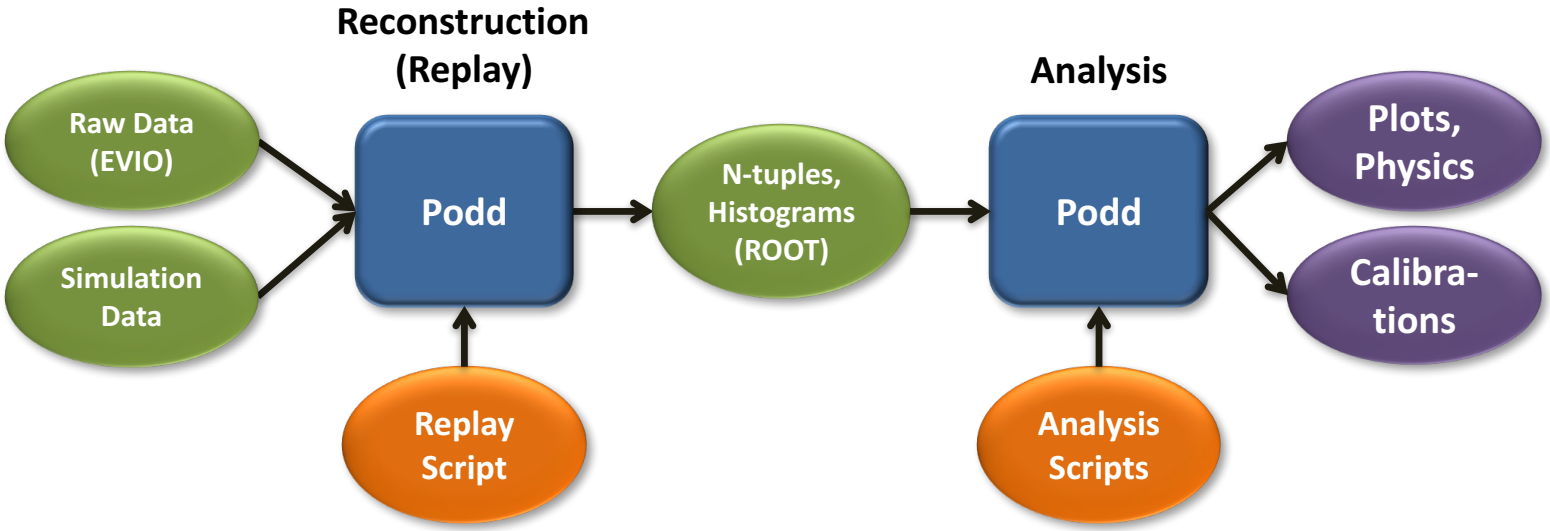
# The Hall A Analyzer ("Podd"): What Does It Do?

- Processes EVIO raw data → ROOT ntuple-style trees + histograms




- Along the way
  - Unpacks (decodes) the raw data
  - Applies calibration constants
  - Reconstructs particle tracks detected in the spectrometers
  - Calculates where these tracks originated at the target (optics)
  - Extracts numbers to help with particle identification (PID)
  - Fetches the info about the incoming beam
  - Computes the physics quantities of each event (kinematics)
  - Applies basic corrections (energy loss, . . . )
- Hopefully, Podd lets you sit back and do physics with ROOT

# The Two Main Steps of a Typical Hall A & C Analysis




1. **Reconstruction (Replay)**
   - Runs in ROOT interpreter (`analyzer` prompt)
   - Calls mostly Podd functions & classes
   - Scripts usually set up by experiment experts or advanced users
   - After setup, usually runs in mass replay on the farm
2. **Analysis**
   - Also runs in ROOT interpreter (`analyzer` prompt)
   - Calls mostly ROOT functions and classes (but needs Podd classes)
   - Done by everyone on the experiment
   - Calibration and final physics usually done here
   - Details depend very much on each experiment

# Podd: Main Features

- Class library on top of ROOT
  - you get all of ROOT for free, plus a lot of extras :)
- Analysis modules available for standard Hall A equipment and more
  - someone else has already done a lot of work for you :)
  - you may get away without writing any code :))
- Everything is a plug-in
  - stuff can be changed easily
  - any code you do write goes into neat modules
- We have a Software Development Kit (SDK)
  - getting started with your own module is really easy
  - you only have to write the non-boring stuff :)
- Plenty of documentation at `http://hallaweb.jlab.org/podd/doc/`
  - but you are here, so you don't even have to read the docs :))

# Podd: More Ways It Makes Your Life Easier

- It's lightweight. No huge dependency blob to download. No missing shared libraries. All you need is ROOT. (Well, almost.) If Podd isn't already installed, it compiles really fast. And it doesn't eat very much memory. (If it does, blame ROOT.)
- It gets configured with simple text files. No complicated SQL database stuff. Nothing to upload, nothing to connect to, no forgotten password troubles. Everything is easy to inspect and edit.
- You control it interactively from the ROOT command prompt. No guesswork what some black box executable really does. Test your setup line-by-line, create an instance of your module, dump variables, print configuration info, all at your convenience.
- It works with the latest ROOT 6. Full C++11 support. Meaningful error messages. No more bizarre script errors.
- Supported on Mac and Linux. Run it right on your laptop.

# Exercise 1: ROOT Warmup in C++11

## C++11 in ROOT: a few simple examples

```
// Initialize a std::vector: initializer list
analyzer [1] vector<double> dvars {3.45, 1.5, 9.91, 6.28, -2.718}
(std::vector<double> &) { 3.45000, 1.50000, 9.91000, 6.28000, -2.71800 }

// Much simpler looping over containers (vectors etc.)
analyzer [2] for( auto x : dvars ) cout << x << ", "; cout << endl;
3.45, 1.5, 9.91, 6.28, -2.718,

// This is old, but still works, now even in ROOT: std::sort
analyzer [3] std::sort(dvars.begin(), dvars.end());
analyzer [4] dvars
(std::vector<double> &) { -2.71800, 1.50000, 3.45000, 6.28000, 9.91000 }

// Define functions on the fly: lambda expressions
analyzer [5] std::sort(dvars.begin(), dvars.end(),
  [](double a, double b) {return b<a;} );

analyzer [6] for( auto x : dvars ) cout << x << ", "; cout << endl;
9.91, 6.28, 3.45, 1.5, -2.718,
```

# Podd: What You (Might) Need To Do

- Tell Podd what detectors and spectrometers you are interested in, where to find data, where to put results, etc. → replay script
- Put databases together
  - Detector channels etc. ("cratemap", "detector maps", get these from your DAQ expert) → `db_cratemap.dat`
  - Geometry, calibration constants, flags → "db files"
- Tell Podd what results ("global variables") you want in the output file → output definition file ("odef")
- (Optional) Define tests (for statistics) and/or cuts (for event selection) → cut definition file ("cdef")
- Get the raw data files
- Find disk space for the output files, which can get large

Your experiment expert may have done much of the above for you :)

# Concepts Important To Know: The Name of the Game

- **Analysis Objects**
  - Any module that produces results
  - Examples: detectors, spectrometers (see later)
  - Each Analysis Object has a unique name
  - Convention for detectors:
    Object name = spectrometer name + "." + detector name
  - Example name: "R.cer": Right HRS ("R") gas Cherenkov ("cer")
- **"Global Variables"**
  - Give access to analysis results
  - Can be a single value or an array of any basic data type
  - Available "globally" (in a global list: `gHaVars`)
  - Each variable has a unique name
  - **Global name = Analysis Object Name + "." + Local Name**
  - Example name: "R.cer.asum_c" (Corrected ADC sum of "R.cer")

# Exercise 2: Listing Global Variables of a Detector

```
[me@centos7 ~]$ cd ~/tutorial
[me@centos7 tutorial]$ source ./setup.sh
[me@centos7 tutorial]$ analyzer
// Create a Cherenkov detector called "R.cer"
analyzer [1] Rcer = new THaCherenkov("R.cer","RHRS gas Cherenkov");

// Initialize it for current time (reads from ~/tutorial/DB database)
analyzer [2] Rcer->Init();

// Now this detector's global variables are set up. Print them
analyzer [3] gHaVars->Print()
 OBJ: THaVar    R.cer.nthit     Number of PMTs with valid TDC
 OBJ: THaVar    R.cer.nahit     Number of PMTs with ADC signal
 OBJ: THaVar    R.cer.t         Raw TDC values
 OBJ: THaVar    R.cer.t_c       Offset-corrected TDC values
 OBJ: THaVar    R.cer.a         Raw ADC values
 OBJ: THaVar    R.cer.a_p       Pedestal-subtracted ADC values
 OBJ: THaVar    R.cer.a_c       Gain-corrected ADC values
 OBJ: THaVar    R.cer.asum_p    Sum of ADC minus pedestal values
 OBJ: THaVar    R.cer.asum_c    Sum of corrected ADC amplitudes
 etc ...
// Don't quit yet ...
```

# Exercise 3: A ROOT Trick: Redirecting Output

The printed list of global variables can get long. It would be nice if we could save it to a file. Easy!

## Redirecting Output

```
// Continuing from the previous exercise
// Tell ROOT to send its output to the file "myvars.txt"
analyzer [4] .> myvars.txt

// Print the variable list again
analyzer [5] gHaVars->Print()

// Note: no output to screen anymore
// Let's turn the output redirection off
analyzer [6] .>

// You can look at the file directly form ROOT. Hit "q" when done.
analyzer [7] .!less myvars.txt

// Again, don't quit yet ...
```

# Exercise 4: Contents of Global Variables

Well, the names and descriptions are interesting, but what values are stored in these variables? Simple enough:

## Contents of Global Variables

```
// Continuing from the previous exercise
// Let's first clear the Cherenkov detector to reset everything
analyzer [8] Rcer->Clear()

// Now print the variable list with the "FULL" option
analyzer [9] gHaVars->Print("FULL")
Collection name='THaVarList', class='THaVarList', size=12
 OBJ: THaVar    R.cer.nthit     Number of PMTs with valid TDC
(Int_t)  0
 OBJ: THaVar    R.cer.nahit     Number of PMTs with ADC signal
(Int_t)  0
 OBJ: THaVar    R.cer.t         Raw TDC values
(Float_t*)[10]  0  0  0  0  0  0  0  0  0  0
 OBJ: THaVar    R.cer.t_c       Offset-corrected TDC values
(Float_t*)[10]  0  0  0  0  0  0  0  0  0  0
 etc ...
// These data would be non-trivial data after analyzing a real event
```

# Exercise 5: Advanced: Individual Global Variables

What if you need to access the data from within a program?

## Individual Global Variables

```
// Fetch a single variable
analyzer [10] tdcs = gHaVars->Find("R.cer.t")
(THaVar *) 0x5153d30
// Non-null return value means the variable was found :)

// How many elements?
analyzer [11] tdcs->GetLen()
(Int_t) 10

// Get a single value
analyzer [12] tdcs->GetValue(0)
(Double_t) 0.00000

// Let's make a mistake
analyzer [13] tdcs->GetValue(10)
Warning in <THaVar::GetValue()>: Whoa! Index out of range,
                                  variable R.cer.t, index 10
(Double_t) 1.00000e+38
// 1e38 is Podd's convention for "invalid number"
```

# Exercise 6: Checking Global Variable Definitions

If in doubt about a global variable, it is best to check how it is defined. For that, you'll have to look at the source code ...

**Method 1: Browse the online documentation (may be outdated)**

Click "Podd" bookmark → Documentation → Analyzer Class Index → THaCherenkov → DefineVariables → DefineVariables (again)

For the impatient:

`http://hallaweb.jlab.org/podd/doc/html_v16/src/THaCherenkov.cxx.html#Do2Vg`

**Method 2: Open the actual source (always up-to-date)**

```
// Open the source file in your favorite editor (even from within Podd)
analyzer [14] .!emacs -nw ~/analyzer/src/THaCherenkov.C
// Search (Ctrl-s) for "DefineVariables"
// You'll see lines like this:
// Varname    Description              C++ variable
   "asum_c", "Sum of corrected ...", "fASUM_c" ,

// Varname    Description              TClonesArray.Class.MemberVariable
   "trx",    "x-position of ...",    "fTrackProj.THaTrackProj.fX" ,
```

# Exercise 6 (cont.): Checking Global Variable Definitions

To understand what "Sum of corrected ADC amplitudes" really means, we need to check how "fASUM_c" is calculated.

## Finding a global variable assignment

```
// With  THaCherenkov still open in your editor, search (Ctrl-s)
// for "fASUM_c". Keep searching until you find an assignment:

   // Copy the data to the local variables.
   if ( adc ) {
     fA[k]   = data;
     fA_p[k] = data - fPed[k];
     fA_c[k] = fA_p[k] * fGain[k];
     // only add channels with signals to the sums
     if( fA_p[k] > 0.0 )
       fASUM_p += fA_p[k];
     if( fA_c[k] > 0.0 )
       fASUM_c += fA_c[k];
     fNAhit++;
   } else {

// Exit (Ctrl-x Ctrl-c)
```

# Types of Analysis Objects

- **Detector**
  - Code/data for analyzing a type of detector.
    Examples: Scintillator, Cherenkov, VDC, BPM
  - Typically embedded in an Apparatus
  - Detectors are not supposed to know about each other
- **Apparatus / Spectrometer**
  - Collection of Detectors
  - Combines data from detectors
  - "Spectrometer": Apparatus with support for tracks
- **Physics Module**
  - Combines data from several apparatuses
  - Typical applications: kinematics calculations, vertex finding, coincidence time extraction
  - Toolbox design: Modules can be chained, combined, used as needed

# What Does a Detector Do?

Detectors provide the following functions to process each event

- All detectors
  - `Clear()`
  - Clears event-by-event data
  - `Decode( event_data )`
  - Retrieves raw data of interest from `event_data`
- "Tracking Detectors"
  - `CoarseTrack( tracks )`
  - Finds tracks without detailed, time-consuming corrections
  - `FineTrack( tracks )`
  - Repeat and/or refine tracking, optionally applying corrections and/or using `CoarseProcess` detector results
- "Non-Tracking Detectors"
  - `CoarseProcess( tracks )`
  - Compute detector response, optionally using coarse tracks (read-only)
  - `FineProcess( tracks )`
  - (Re)compute detector response, optionally using fine tracks and/or target quantities

# What Does a Spectrometer Do?

1. For all tracking detectors
   - `CoarseTrack( tracks )`
   - Finds tracks without detailed, time-consuming corrections
2. For all non-tracking detectors (*e.g.* PID detectors)
   - `CoarseProcess( tracks )`
   - Compute detector response, optionally using coarse tracks (read-only)
3. For all tracking detectors
   - `FineTrack( tracks )`
   - Repeat and/or refine tracking, optionally applying corrections and/or using `CoarseProcess` detector results
4. Reconstruct tracks to target
   - `FindVertices()`
5. For all non-tracking detectors
   - `FineProcess( tracks )`
   - (Re)compute detector response, optionally using fine tracks and/or target quantities
6. Compute additional attributes of tracks (*e.g.* momentum, beta, "Golden Track")
   - `TrackCalc()`
7. Combine all PID detectors to get overall PID for each track
   - `CalcPID()`

# Exercise 7: Taking a Look at Spectrometer Processing

Open ~/analyzer/src/THaSpectrometer.C in an editor (or on web)
Search for "::Reconstruct". Scroll down. You'll see (DEBUG code cut):

```
  // Do prior analysis stages if not done yet
  if( !IsDone(kTracking))
    Track(); //Calls CoarseProcess, CoarseTrack, FineTrack, FindVertices

  // Fine processing.  Pass the precise tracks to the
  // remaining detectors for any precision processing.
  // PID likelihoods should be calculated here.
  TIter next( fNonTrackingDetectors );
  while( THaNonTrackingDetector* theNonTrackDetector =
         static_cast<THaNonTrackingDetector*>( next() )) {
    theNonTrackDetector->FineProcess( *fTracks );
  }

  // Compute additional track properties (e.g. beta)
  // Find "Golden Track" if appropriate.
  TrackCalc();

  // Compute combined PID
  if( fPID ) CalcPID();
```

# Exercise 8: Creating a Spectrometer Object w/Detectors

Cool! Let's set up a spectrometer module full of detectors we want to analyze

Restart Podd before doing this exercise to clear old objects out.

```
// Create a Right HRS spectrometer called "R"
analyzer [1] RHRS = new THaHRS("R","Right HRS");
// Add a bunch of detectors, including our old Cherenkov friend
analyzer [2] RHRS->AddDetector( new THaVDC("vdc","RHRS VDC"));
analyzer [3] RHRS->AddDetector( new THaScintillator("s0","RHRS S0"));
analyzer [4] RHRS->AddDetector( new THaScintillator("s2","RHRS S2m"));
analyzer [5] RHRS->AddDetector( new THaCherenkov("cer","RHRS C'kov"));
// Check: Print the detector list of this apparatus
analyzer [6] RHRS->GetDetectors()->Print()
Collection name='TList', class='TList', size=4
 AOBJ: THaVDC           vdc     ""      RHRS VDC
 AOBJ: THaScintillator  s0      ""      RHRS S0
 AOBJ: THaScintillator  s2      ""      RHRS S2m
 AOBJ: THaCherenkov     cer     ""      RHRS gas Cherenkov
// Init the spectrometer (for Mar 6, 2016, 00:00h)
analyzer [7] RHRS->Init( TDatime(2016,3,6,0,0,0) );
analyzer [8] gHaVars->GetSize()
(Int_t) 213
```

# Exercise 9: More About Detectors Within Apparatuses

```
// Let's look at the entire initialized RHRS object including detectors.
// Note the 4th column is set to the "name" each module will use
analyzer [9] RHRS->Print("DETS")
AOBJ: THaHRS    R       "R."    Rights HRS
Collection name='TList', class='TList', size=4
 AOBJ: THaVDC           vdc     "R.vdc."        RHRS VDC
 AOBJ: THaScintillator  s0      "R.s0."         RHRS S0
 AOBJ: THaScintillator  s2      "R.s2."         RHRS S2m
 AOBJ: THaCherenkov     cer     "R.cer."        RHRS gas Cherenkov

// Getting or setting parameters of a detector is easy.
// Setting is best done  before initializing the apparatus, though
analyzer [10] RHRS->GetDetector("cer")->SetDebug(2)

// To call class-specific functions, you need a cast:
analyzer [11] THaVDC* vdc = (THaVDC*)RHRS->GetDetector("vdc")
analyzer [12] vdc->GetVDCAngle()
(Double_t) -0.787693

// Podd tries to use units of "m", "rad" and "GeV" consistently
analyzer [21] TMath::RadToDeg() * vdc->GetVDCAngle()
(double) -45.1315
```

# The Event Loop

Now that we have our nifty `RHRS` object, what do we do with it?

Somebody has to call the various processing functions in the right order.

Somebody has to handle the input and output.

Somebody has to coordinate the initialization.

This is the job of the event loop class: `THaAnalyzer`

# Global Lists

Podd keeps global lists of analysis-related objects. They are accessible from the interpreter command line. You will need to interact with some of them.

`THaAnalyzer` takes these lists as input.

- `gHaVars`
  - Home of the Global Variables. We've already covered them :)
  - You actually don't need to access this list directly very often
- `gHaApps`
  - Apparatuses (spectrometers, beam) that you want to analyze
  - You need to fill this list in your replay script before any real action can start
  - Apparatuses are processed in the order in which they appear here
  - The apparatuses contain the detectors, so there is no separate list of detectors
- `gHaPhysics`
  - Physics Modules live here
  - Like `gHaApps`, fill this list with modules before starting any processing

# Exercise 10: Running an Analysis from the Command Line

```
// Add our fresh RHRS object to the list of apparatuses to be analyzed
// Note: the object does not have to be initialized
analyzer [9] gHaApps->Add(RHRS);

// Make an analyzer object. Only one is allowed per session (singleton)
analyzer [10] analyzer = new THaAnalyzer;

// Make a "run" object that gives access to a CODA file
analyzer [11] run = new THaRun("/data/raw/gmp_23062.dat.0");

// Let's limit ourselves to 1000 events in this run
analyzer [12] run->SetLastEvent(1000);

// Tell the analyzer where to write the output ROOT file
analyzer [14] analyzer->SetOutFile("rootfiles/junk.root");

// And off we go ...
analyzer [15] analyzer->Process(run)
```

# Exercise 11: Inspecting Analysis Results Right Away

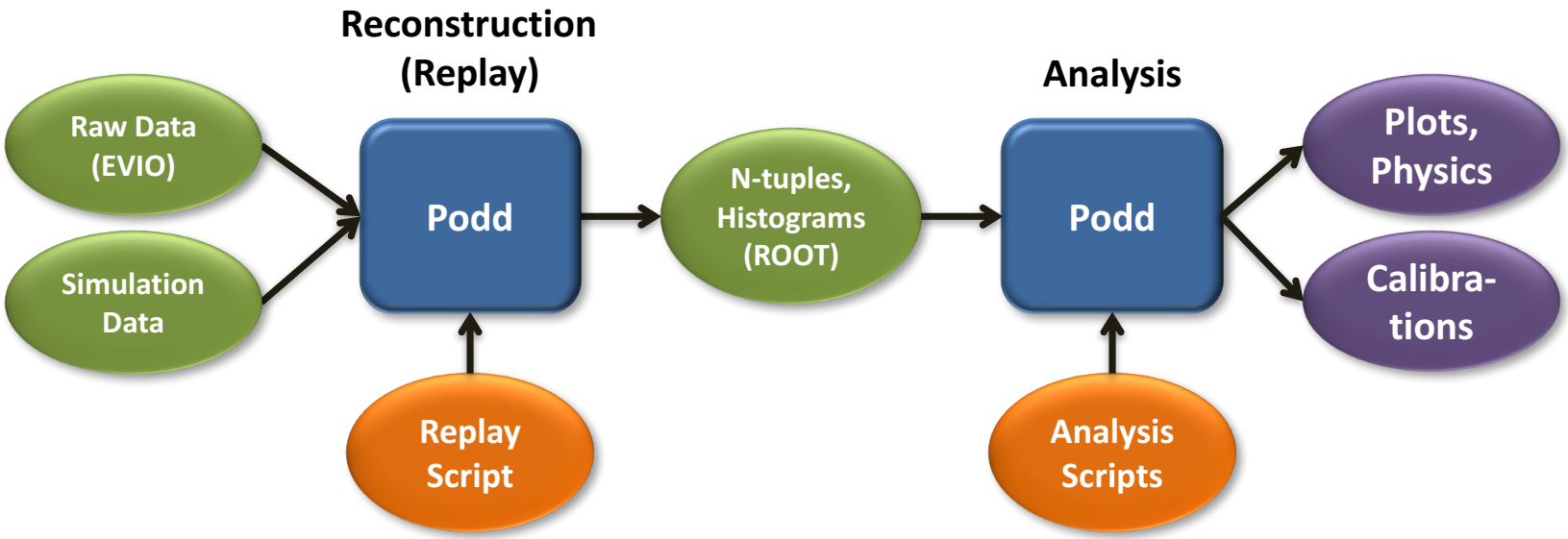
After `THaAnalyzer` completes, the output ROOT file is still open. The ROOT tree is in memory, and we can inspect the results right away.

```
// List what's in ROOT's memory
analyzer [16] .ls
TFile**         rootfiles/junk.root
 TFile*         rootfiles/junk.root
  OBJ: TTree    T       Hall A Analyzer Output DST : 0 at: 0x38404c0
  KEY: THaRun   Run_Data;2
  KEY: TTree    T;1     Hall A Analyzer Output DST

// Two items of interest:
// (1) The ROOT Tree "T". Let's list its contents:
analyzer [17] T->Print()
...
// Hmm ... not much there. But wait, there was a warning during replay.
// We only have event headers in this file! Well, at least something:
analyzer [18] T->Draw("fEvtHdr.fEvtLen")

// (2) A copy of our (initialized) run object "Run_Data". Useful info :)
analyzer [19] Run_Data->Print()
...
```

# Getting Our Bearings: Where Are We Again?




- Reconstruction (Replay)
  - `Analyzer->Process(run)`
- Analysis
  - `T->Draw(...)`
  - `Run_Data->Print()`

# Output Definitions

- Choose "global variables" to include in ROOT output tree
- Tree branches can be dynamically defined for each replay via input file

## Example Output Definition File

```
# A single variable: "status" of the RHRS
variable R.status
# A wildcard expression: all variables from the GoldenTrack module
block R.gold.*
# All RHRS track data (focal plane as well as at target)
# (this is overkill, should narrow it down)
block R.tr.*
```

- Much more possible
  - Arithmetic expressions
  - Defining and appying cuts
  - 1D and 2D histograms
  - EPICS variables
  - Scalers
- Full documentation on the web ▸ docs (Bob Michaels)

# Exercise 12: Adding Results to the Output

We need to redo the previous exercise with an output definition file.
Restart the analyzer before this exercise.

```
// To save a lot of retyping, there is a script to recreate the setup
// from the previous exercise:
[me@centos7 ~]$ cd ~/tutorial
// First, let's peek at the script and .odef file
[me@centos7 tutorial]$ less replay/toy_setup.C
[me@centos7 tutorial]$ less replay/toy_replay.odef
[me@centos7 tutorial]$ analyzer
analyzer [0] .x replay/toy_setup.C
analyzer [1] analyzer->SetOdefFile("replay/toy_replay.odef")
analyzer [2] analyzer->Process(run)
...
analyzer [3] T->Print()
// Wow, lots of information. Now you can go explore. Some examples:
// Detector diagnostics
analyzer [3] T->Draw("R.s2.la")
analyzer [4] T->Draw("R.s2.la_c")
analyzer [5] T->Draw("R.s2.trpad")
analyzer [6] T->Draw("R.vdc.v2.wire")
// Reconstructed track parameters: target "y" (projection of z-beam)
analyzer [7] T->Draw("R.tr.tg_y>>hy(250,-.15,.15)")
```

# Database

Anyone else think this was too good to be true?
Well, I cheated. I had the databases all set up for you.

## Example Database File ~/tutorial/DB/20160205/db_R.cer.dat

```
----[ 2016-02-05 00:00:00 -0500 ]
R.cer.detmap =
      1    20    32    41     1  1881
      2    11    32    41     1  1877
R.cer.npmt = 10
R.cer.position = 0  0  1.99
R.cer.size = 1  0.4  1
R.cer.tdc.offsets = 0  0  0  0  0  0  0  0  0  0
R.cer.adc.pedestals = 439.3  383.5  352.2  492.7  557.1  553  563.1  489.4  227.2  465.6
R.cer.adc.gains = 1.06  0.92  1.08  1.05  0.99  0.99  1  1.01  1.01  0.97

----[ 2016-09-10 00:00:00 -0400 ]
R.cer.position = -0.08  -0.008  1.8
R.cer.size = 1.22  0.302  1.37
R.cer.adc.pedestals = 439.8  384.3  352.8  493.1  557.1  553.2  564.1  490  227.3  465.9
R.cer.adc.gains = 0.926  0.919  1.139  1.002  0.95  0.997  0.989  1.014  1.05  0.983
```

- Flat text files
- Key/value pairs with support for scalars, arrays, matrices, strings
- Support for incremental validity periods and time zones
- Strongly recommend to keep files under version control

# Database

Anyone else think this was too good to be true?
Well, I cheated. I had the databases all set up for you.

## Example Database File ~/tutorial/DB/20160205/db_R.cer.dat

```
----[ 2016-02-05 00:00:00 -0500 ]
R.cer.detmap =
      1    20    32    41     1  1881
      2    11    32    41     1  1877
R.cer.npmt = 10
R.cer.position = 0  0  1.99
R.cer.size = 1  0.4  1
R.cer.tdc.offsets = 0  0  0  0  0  0  0  0  0  0
R.cer.adc.pedestals = 439.3  383.5  352.2  492.7  557.1  553  563.1  489.4  227.2  465.6
R.cer.adc.gains = 1.06  0.92  1.08  1.05  0.99  0.99  1  1.01  1.01  0.97

----[ 2016-09-10 00:00:00 -0400 ]
R.cer.position = -0.08  -0.003  1.8
R.cer.size = 1.22  0.302  1.37
R.cer.adc.pedestals = 439.8  384.3  352.8  493.1  557.1  553.2  564.1  490  227.3  465.9
R.cer.adc.gains = 0.926  0.919  1.139  1.002  0.95  0.997  0.989  1.014  1.05  0.983
```

- Flat text files
- Key/value pairs with support for scalars, arrays, matrices, strings
- Support for incremental validity periods and time zones
- Strongly recommend to keep files under version control

# Exercise 13: Database Conversion

Podd 1.5 an earlier used fixed-format database files for many detectors. With Podd 1.6, all modules have been switched the key-value format. Old databases can be converted with the `dbconvert` utility.

```
[me@centos7 ~]$ cd ~/database
// Take a look at the original database files (version 1.5)
[me@centos7 database]$ ls DB_gmp12_Ar40_v15
// Now run the conversion program (all one line)
// We'll create new subdirectories, one for each run apparent period
[me@centos7 database]$ dbconvert --subdirs 20140213,20150324,20160205,
  20170130 DB_gmp12_Ar40_v15 DBtest 2>&1 | tee convert.log
// Check which detectors failed to convert
[me@centos7 database]$ grep "Failed to read" convert.log
// Check if there is any information why a certain conversion failed
[me@centos7 database]$ grep -B1 DB_gmp12_Ar40_v15/20140213/db_L.a1.dat
  convert.log
Error in <Cherenkov::ReadDB(file="...")>: Database inconsistency.
Defined 52 channels in detector map, but have 48 total channels
  (24 mirrors with 1 ADC and 1 TDC each)
// Now you need to fix the source by hand - it's really incorrect
```

# Exercise 14: Optics Run Replay

As a final exercise, let's do a more full-fledged replay and analysis of an older optics calibration run.
(g2p run 3132, 12 Mar 2012, LHRS @ 2.228 GeV, 6°, thin carbon foil, sieve slit, septum)

## Optics Run Replay

```
[me@centos7 ~]$ cd ~/tutorial/replay
[me@centos7 replay]$ analyzer
analyzer [0] .x replay.C
Here are the data files:
g2p_3132.dat.0  gmp_22788.dat.0  gmp_23062.dat.0  shms_all_00484.dat
Run number? 3132
Number of events to replay (-1=all)? -1
...
314292  events read
204415  events accepted
Physics_master   GoodGoldenTrack      313476     203599 (64.9%)
```

# Exercise 15: Optics Run Analysis

## Optics Run Analysis

```
// Continuing from previous session
// Or do analyzer /data/ROOTfiles/g2p_3132.dat
// Check track target-y and vertex-z. (Very forward scattering 6 deg.)
analyzer [1] T->Draw("L.tr.tg_y>>hy(250,-.15,.15)")
analyzer [2] T->Draw("L.vx.z>>hz(250,-.3,.3)")
// 12C(e,e') Momentum distribution
analyzer [3] T->Draw("L.gold.dp>>hdp(500,-0.015,0.015)")
// Spectrometer angular acceptance with cut on elastic peak
analyzer [4] TCut cutE("cutE","abs(L.gold.dp-0.0095)<0.0015&&
  L.cer.asum_c>500")
analyzer [5] TH2F* h2p = new TH2F("h2p","th vs ph",1000,-.05,.05,
  1000,-.1,.1);
analyzer [6] T->Draw("L.gold.th:L.gold.ph>>h2p",
  "abs(L.gold.th)<.1&&abs(L.gold.ph)<.1&&abs(L.gold.dp-0.0095)<0.0015",
  "COLZ");
// Invariant mass (12C target)
analyzer [7] T->Draw("L.ekine.W2>>hW2(250,120,130)",cutE)
analyzer [8] TMath::Sq(12*0.931494)
(Double_t) 124.946
// Reaction Q^2 vs. scattering angle (deviation from central angle)
analyzer [9] T->Draw("L.ekine.Q2:L.gold.ph>>
  hQ2ph(500,-.03,0.03,500,0.02,0.08)",cutE,"COLZ")
```

# Status

- Stable version: 1.5.37 (03-Mar-2017) ▸ web
  - Bugfixes
  - 1.5.x releases are binary-compatible
- Development version: 1.6-beta3 (19-Jan-2017) ▸ web
  - New database format
  - Many new features (see next), not all fully implemented/tested yet.
  - Hope to finalize by summer 2017 for fall run
  - Preliminary Release Notes available ▸ web
- Repository ▸ GitHub
  - For experts. Things may change unexpectedly.
  - Download:

    ```
    git clone https://github.com/JeffersonLab/analyzer.git
    ```

# Version 1.6

- Completed
  - Modular decoder (Bob Michaels)
  - Simulation event data decoder API
  - Miscellaneous
    - ★ Improved formula & test package (removed limitations)
    - ★ `scons` build system
    - ★ Rewritten, modular hardware channel decoder (`THaDecData`)
    - ★ EVIO from external library
    - ★ Many small code improvements
- In Progress
  - Generalized database interface
  - Improved VDC track reconstruction

# VDC Algorithm Improvements

Version 1.5.38

- Disallow UV ambiguities (configurable)
- UV fiducial cut
- Proper lower-upper matching cut
- Disallow cluster sharing

→ Guarantees clean single track at expense of slightly lower tracking efficiency

Version 1.6

- Cluster shape analysis
- Overlapping cluster splitting (to do)
- 3-parameter cluster fit (to do)
- Cluster $t_0$ cut
- UV fiducial cut
- Proper lower-upper matching cut
- Disallow cluster sharing
- Old VDC code for reference (to do)

→ Allows multi-tracks, improves tracking efficiency, high-rate capable

# Resources

- Web site ▸ home page
  - Documentation
  - Release Notes
  - Software Development Kit (SDK)
  - Source code downloads
  - Archived tutorials & example replays
- Issue & task tracker (Redmine) ▸ Redmine
- Bi-weekly Hall A/C software meeting: Wednesdays, 11am, L201
- Mailing list: `halla_software@jlab.org`. Subscribe on ▸ mailman
- Analysis Workshop archive ▸ archive (includes older tutorials)

Thanks!