ROOT, AN OBJECT ORIENTED DATA ANALYSIS FRAMEWORK

René Brun, Fons Rademakers, Suzanne Panacek
CERN, Geneva, Switzerland

Abstract
ROOT is an *object-oriented framework* aimed at solving the data analysis challenges of high-energy physics. Here we discuss the main components of the framework. We begin with an overview describing the framework's organization, the interpreter CINT, its automatic interface to the compiler and linker ACLiC, and an example of a first interactive session. The subsequent sections cover histogramming and fitting. Then, ROOT's solution to storing and retrieving HEP data, building and managing of ROOT files, and designing ROOT trees. Followed by a description of the collection classes, the GUI classes, how to add your own classes to ROOT, and PROOF, ROOT's parallel processing facility.

1 INTRODUCTION

In the mid 1990's, the designers of ROOT had many years of experience developing interactive data analysis tools and simulation packages. They had lead successful projects such as PAW, PIAF, and GEANT, and they knew the twenty-year-old FORTRAN libraries had reached their limits. Although still very popular, these tools could not scale up to the challenges offered by the Large Hadron Collider, where the data is a few orders of magnitude larger. At the same time, computer science had made leaps of progress especially in the area of Object Oriented Design, and the time had come to take advantage of it.

The first version or ROOT, version 0.5, was released in 1995, and version 1.0 was released in 1997. Since then it has been released early and frequently to expose it to thousands of eager users to pound on, report bugs, and contribute possible fixes. More users find more bugs, because more users add different ways of stressing the program. By now, after six years, many users have stressed ROOT in many ways, and it is quiet mature.

ROOT is an *object-oriented framework*. A *framework* is a collection of cooperating classes that make up a reusable solution for a given problem. The two main differences between frameworks and class libraries are:

Behavior versus Protocol: A class library is a collection of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the rules for behaviors that can be combined.

Implementation versus Design: With class libraries programmers reuse only implementations, whereas with frameworks they reuse design. A framework embodies the way a family of related classes work together.

Object-Oriented Programming offers considerable benefits compared to Procedure-Oriented Programming:

Encapsulation enforces data abstraction and increases opportunity for reuse.

Sub classing and inheritance make it possible to extend and modify objects.

Class hierarchies and containment hierarchies provide a flexible mechanism for modeling real-world objects and the relationships among them.

Complexity is reduced because there is little growth of the global state, the state is contained within each object, rather than scattered through the program in the form of global variables.

Objects may come and go, but the basic structure of the program remains relatively static, increases opportunity for reuse of design.

1.1 Main Components of ROOT

- A hierarchical object-oriented database (machine independent, highly compressed, supporting schema evolution and object versioning)
- A C++ interpreter
- Advanced statistical analysis tools (classes for multi-dimensional histogramming, fitting and minimization)
- Visualization tools (classes for 2D and 3D graphics including an OpenGL interface)
- Advanced query mechanisms to select information in very large data sets (ROOT Trees)
- A rich set of container classes that are fully I/O aware (list, sorted list, map, btree, hashtable, object array, etc.)
- An extensive set of GUI classes (windows, buttons, combo-box, tabs, menus, item lists, icon box, tool bar, status bar and many more)
- An automatic HTML documentation generation facility
- Run-time object inspection capabilities
- Client/server networking classes
- Shared memory support
- Multi-threading support
- Remote database access either via a special daemon or via the Apache web server
- Ported to all known Unix and Linux systems and also to Windows 95 and NT

1.2 The Organization of the ROOT Framework

The ROOT framework has about 460 classes grouped by functionality into shared libraries. The libraries are designed and organized to minimize dependencies, such that you can include just enough code for the task at hand rather than having to include all libraries or one monolithic chunk.

The core library (`libCore.so`) contains the essentials; it needs to be included for all ROOT applications. `libCore` is made up of Base classes, Container classes, Meta information classes (for RTTI), Networking classes, Operating system specific classes, and the ZIP algorithm used for compression of the ROOT files.

The CINT library (`libCint.so`) is also needed in all ROOT applications, but `libCint` can be used independently of `libCore`, in case one only needs the C++ interpreter and not ROOT.

Figure 1shows the libraries and their dependencies. For example, a batch program, one that does not have a graphic display, which creates, fills, and saves histograms and trees, only needs the core (`libCore` and `libCint`), `libHist` and `libTree`. If other libraries are needed ROOT loads them dynamically. For example if the `TreeViewer` is used, `libTreePlayer` and all the libraries the `TreePlayer` box below has an arrow to, are loaded also. In this case: `GPad`, `Graf3d`, `Graf`, `HistPainter`, `Hist`, and `Tree`. The difference between `libHist` and `libHistPainter` is that the former needs to be explicitly linked and the latter will be loaded automatically at runtime when needed. In the diagram, the dark boxes outside of the core are automatically loaded libraries, and the light colored ones are not automatic. Of course, if one wants to access an automatic library directly, it has to be explicitly linked also.

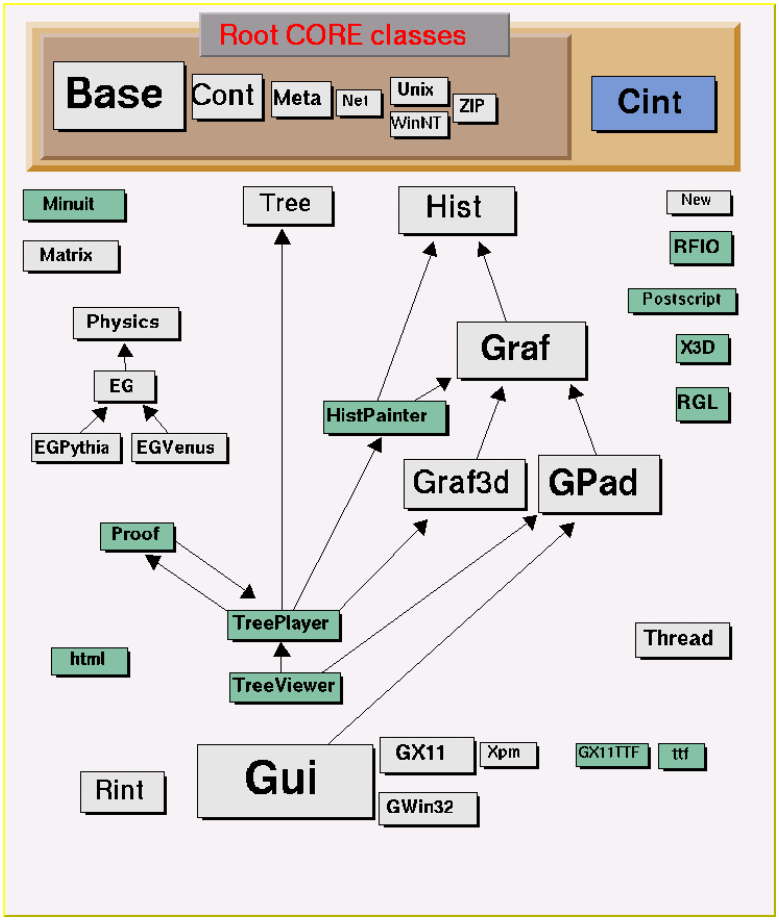

Figure 1: The ROOT libraries and their dependencies

Here is a short description for each library, the ones marked with a * are only installed when the options specified them.

- libCint.so is the C++ interpreter (CINT).
- libCore.so is the Base classes.
- libEG.so is the abstract event generator interface classes.
- *libEGPythia.so io the Pythia5 event generator interface.
- *libEGPythia6.so is the Pythia6 event generator interface.
- libEGVenus.so is the Venus event generator interface.
- libGpad.so is the pad and canvas classes which depend on low level graphics.
- libGraf.so is the 2D graphics primitives (can be used independent of libGpad.so).
- libGraf3d.so is the3D graphics primitives.
- libGui.so is the GUI classes (depends on low level graphics).
- libGX11.so is the low level graphics interface to the X11 system.
- *libGX11TTF.so is an add on library to libGX11.so providing TrueType fonts.
- libHist.so is the histogram classes.
- libHistPainter.so is the histogram painting classes.
- libHtml.so is the HTML documentation generation system.

- libMatrix.so is the matrix and vector manipulation.
- libMinuit.so - The MINUIT fitter.
- libNew.so is the special global new/delete, provides extra memory checking and interface for shared memory (optional).
- libPhysics.so is the physics quantity manipulation classes (TLorentzVector, etc.).
- libPostscript.so is the PostScript interface.
- libProof.so is the parallel ROOT Facility classes.
- *libRFIO.so is the interface to CERN RFIO remote I/O system.
- *libRGL.so is the interface to OpenGL.
- libRint.so is the interactive interface to ROOT (provides command prompt).
- *libThread.so is the Thread classes.
- libTree.so is the TTree object container system.
- libTreePlayer.so is the TTree drawing classes.
- libTreeViewer.so is the graphical TTree query interface.
- libX3d.so is the X3D system used for fast 3D display.

1.3 CINT: The C/C++ Interpreter

A key component of the ROOT framework is the CINT C/C++ interpreter. CINT, written by Masaharu Goto of Hewlett Packard Japan, covers 95% of ANSI C and about 85% of C++ (template support is being worked on, exceptions are still missing). CINT is complete enough to be able to interpret its own 70,000 lines of C and to let the interpreted interpreter interpret a small program.

The advantage of a C/C++ interpreter is that it allows for fast prototyping since it eliminates the typical time-consuming edit/compile/link cycle. Once a script or program is finished, you can compile it with a standard C/C++ compiler (gcc) to machine code and enjoy full machine performance. Since CINT is very efficient (for example, for/while loops are byte-code compiled on the fly), it is quite possible to run small programs in the interpreter. In most cases, CINT out performs other interpreters like Perl and Python.

Existing C and C++ libraries can easily be interfaced to the interpreter. This is done by generating a dictionary from the function and class definitions. The dictionary provides CINT with all necessary information to be able to call functions, to create objects and to call member functions. A dictionary is easily generated by the program `rootcint` that uses as input the library header files and produces as output a C++ file containing the dictionary. You compile the dictionary and link it with the library code into a single shared library. At run time, you dynamically link the shared library, and then you can call the library code via the interpreter. This can be a very convenient way to quickly test some specific library functions. Instead of having to write a small test program, you just call the functions directly from the interpreter prompt.

The CINT interpreter is fully embedded in the ROOT system. It allows the ROOT command line, scripting and programming languages to be identical. The embedded interpreter dictionaries provide the necessary information to automatically create GUI elements like context pop-up menus unique for each class and for the generation of fully hyperized HTML class documentation. Further, the dictionary information provides complete run-time type information (RTTI) and run-time object introspection capabilities.

On the ROOT command line, you can enter C++ statements for CINT to interpret. You can also write and edit a script and tell CINT to execute the statements in it with the .x command:

```
root[ ] .x MyScript.C
```

1.4 ACLiC: The Automatic Compiler of Libraries for CINT

Instead of having CINT interpret your script there is a way to have your scripts compiled, linked and dynamically loaded using the C++ compiler and linker. The advantage of this is that your scripts will run with the speed of compiled C++ and that you can use language constructs that are not fully supported by CINT. On the other hand, you cannot use any CINT shortcuts and for small scripts, the overhead of the compile/link cycle might be larger than just executing the script in the interpreter.

ACLiC will build a CINT dictionary and a shared library from your C++ script, using the compiler and the compiler options that were used to compile the ROOT executable. You do not have to write a makefile remembering the correct compiler options, and you do not have to exit ROOT.

To build, load, and execute a script with ACLiC you append a "++" at the end of the script file name, and use the CINT .x command.

```
root[] .x MyScript.C++
```

ACLiC executes two steps and a third one if needed. These are:

1. Calling `rootcint` to create a CINT dictionary.
2. Calling the compiler to build the shared library from the script
3. If there are errors, it calls the compiler to build a dummy executable to clearly report unresolved symbols.

1.5 First Interactive Session

In this first session, start the ROOT interactive program. This program gives access via a command-line prompt to all available ROOT classes. By typing C++ statements at the prompt, you can create objects, call functions, execute scripts, etc. Go to the directory `$ROOTSYS/tutorials` and type:

```
bash$ root
root [0] 1+sqrt(9)
(double)4.000000000000e+00
root [1] for (int i = 0; i < 5; i++) printf("Hello %d\n", i)
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
root [2] .q
bash $
```

As you can see, if you know C or C++, you can use ROOT, and there is no new command-line or scripting language to learn. To exit use `.q`, which is one of the few "raw" interpreter commands. The dot is the interpreter escape symbol. There are also some dot commands to debug scripts (step, step over, set breakpoint, etc.) or to load and execute scripts. Let's now try something more interesting. Again, start root:

```
bash$ root
root [0] TF1 f1("func1","sin(x)/x", 0, 10)
root [1] f1.Draw()
root [2] f1.Integral(0,2)
root [3] f1.Dump()
root [4] .q
```

Here you create an object of class `TF1`, a one-dimensional function. In the constructor, you specify a name for the object (which is used if the object is stored in a database), the function and the upper and lower value of x. After having created the function object you can, for example, draw the object by executing the **`TF1::Draw()`** member function.

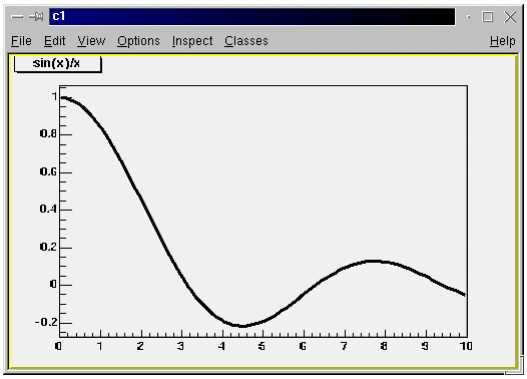

Figure 2: The result of TF1:Draw

Now, move the mouse over the picture and see how the shape of the cursor changes whenever you cross an object. At any point, you can press the right mouse button to pop-up a context menu showing the available member functions for the current object. For example, move the cursor over the function so that it becomes a pointing finger, and then press the right button. The context menu shows the class and name of the object. Select item `SetRange` and put 10,10 in the dialog box fields. (This is equivalent to executing the member function `f1.SetRange(10,10)` from the command-line prompt, followed by `f1.Draw()`.) Using the `Dump()` member function (that each ROOT class inherits from the basic ROOT class `TObject`), you can see the complete state of the current object in memory. The `Integral()` function shows the function integral between the specified limits.

1.6 The Object Browser

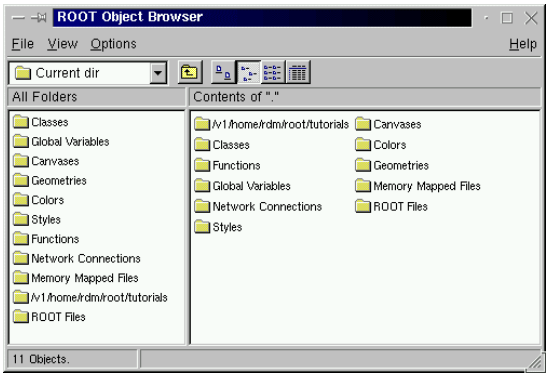

Figure 3: the Object Browser

Using the ROOT Object Browser all objects in the ROOT framework can be browsed and inspected. To create a browser object, type:

```
root [] new TBrowser
```

The browser displays in the left pane the ROOT collections and in the right pane the objects in the selected collection. Double clicking on an object will execute a default action associated with the class of the object. Double clicking on a histogram object will draw the histogram. Right clicking on an object will bring up a context menu (just as in a canvas).

2 HISTOGRAMS AND FITS

Let's start ROOT again and run the following two scripts. Note: if the above doesn't work, make sure you are in the tutorials directory.

```
bash$ root
root [] .x hsimple.C
root [] .x ntuple1.C
 // interact with the pictures in the canvas
root [] .q
```

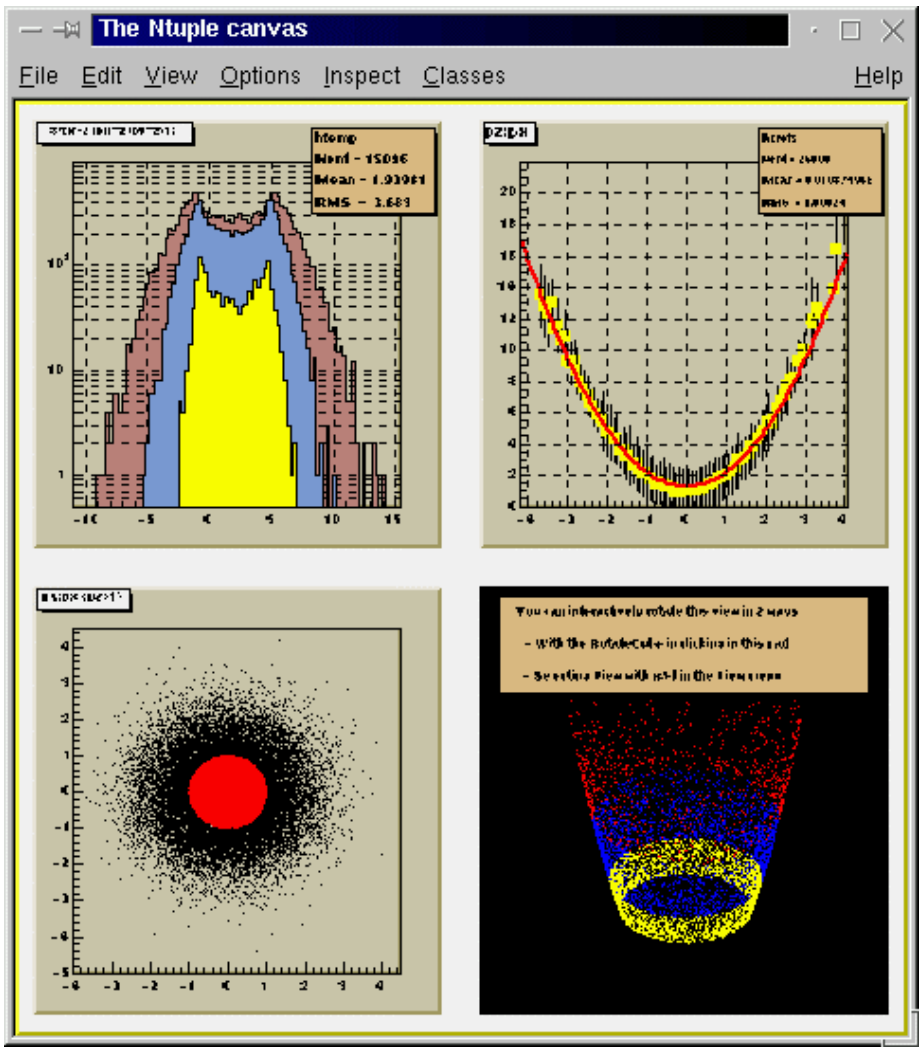

Figure 4: ROOT Histograms

Script `hsimple.C` (see `$ROOTSYS/tutorials/hsimple.C`) creates some 1D and 2D histograms and an Ntuple object. (An Ntuple is a collection of tuples; a tuple is a set of numbers.) The histograms and Ntuple are filled with random numbers by executing a loop 25,000 times. During the filling the 1D histogram is drawn in a canvas and updated each 1,000 fills. At the end of the script, the histogram and Ntuple objects are stored in a ROOT database.

The `ntuple1.C` script uses the database created in the previous script. It creates a canvas object and four graphics pads. In each of the four pads a distribution of different Ntuple quantities is drawn. Typically, data analysis is done by drawing in a histogram with one of the tuple quantities when some of the other quantities pass a certain condition. For example, our Ntuple contains the quantities `px`, `py`, `pz`,

random and `i`. This command will fill a histogram containing the distribution of the `px` values for all tuples for which `pz < 1`.

```
root[] ntuple->Draw("px", "pz< 1")
```

2.1 The Histogram Classes

ROOT supports the following histogram types:

1-D histograms:

- TH1C : histograms with one byte per channel. Maximum bin content = 255
- TH1S : histograms with one short per channel. Maximum bin content = 65535
- TH1F : histograms with one float per channel. Maximum precision 7 digits
- TH1D : histograms with one double per channel. Maximum precision 14 digits

2-D histograms:

- TH2C : histograms with one byte per channel. Maximum bin content = 255
- TH2S : histograms with one short per channel. Maximum bin content = 65535
- TH2F : histograms with one float per channel. Maximum precision 7 digits
- TH2D : histograms with one double per channel. Maximum precision 14 digits

3-D histograms:

- TH3C : histograms with one byte per channel. Maximum bin content = 255
- TH3S : histograms with one short per channel. Maximum bin content = 65535
- TH3F : histograms with one float per channel. Maximum precision 7 digits
- TH3D : histograms with one double per channel. Maximum precision 14 digits

Profile histograms: (`TProfile` and `TProfile2D`)

Profile histograms are used to display the mean value of Y and its RMS for each bin in X. Profile histograms are in many cases an elegant replacement of two-dimensional histograms. The inter-relation of two measured quantities X and Y can always be visualized by two-dimensional histogram or scatter-plot; If Y is an unknown (but single-valued) approximate function of X, this function is displayed by a profile histogram with much better precision than by a scatter-plot.

All histogram classes are derived from the base class TH1.

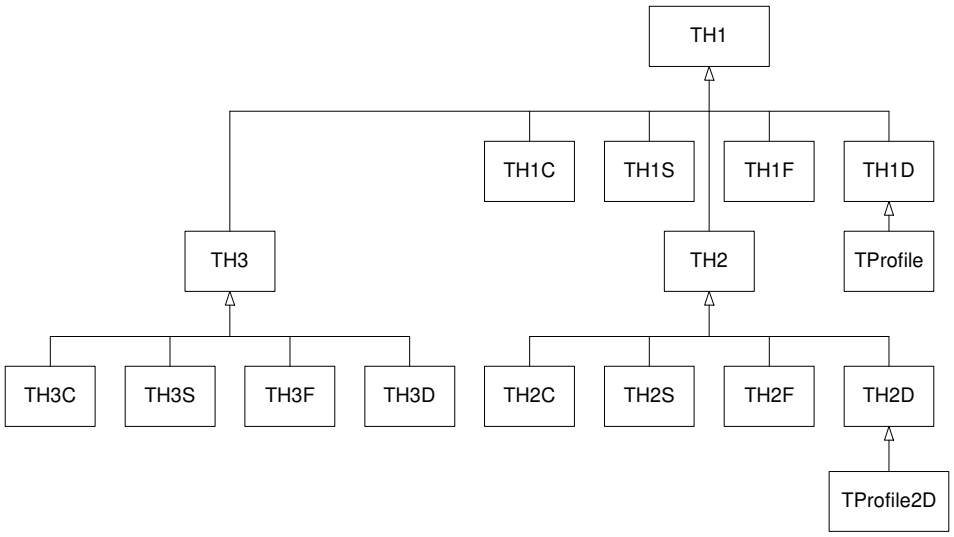

Figure 5: The Histogram Classes

The TH*C classes also inherit from the array class `TArrayC`.
The TH*S classes also inherit from the array class `TArrayS`.
The TH*F classes also inherit from the array class `TArrayF`.
The TH*D classes also inherit from the array class `TArrayD`.

2.2 Creating histograms

Histograms are created by invoking one of the constructors:

```
TH1F *h1 = new TH1F("h1","h1 title",100,0,4.4);
TH2F *h2 = new TH2F("h2","h2 title",40,0,4,30,-3,3);
```

Histograms may also be created by:

Calling the `Clone` method, see below
Making a projection from a 2-D or 3-D histogram, see below
Reading a histogram from a file

2.3 Fixed or variable bin size

All histogram types support either fixed or variable bin sizes. 2-D histograms may have fixed size bins along X and variable size bins along Y or vice-versa. The functions to fill, manipulate, draw or access histograms are identical in both cases. Each histogram always contains three objects `TAxis`: `fXaxis`, `fYaxis`, and `fZaxis`. To access the axis parameters, do:

```
TAxis *xaxis = h->GetXaxis();
Double_t binCenter = xaxis->GetBinCenter(bin);
```

See class `TAxis` for a description of all the access functions. The axis range is always stored internally in double precision.

2.4 Bin numbering convention

For all histogram types: nbins, xlow, xup

bin = 0;	underflow bin
bin = 1;	first bin with low-edge xlow INCLUDED
bin = nbins;	last bin with upper-edge xup EXCLUDED
bin = nbins+1;	overflow bin

In case of 2-D or 3-D histograms, a "global bin" number is defined. For example, assuming a 3-D histogram with binx, biny, binz, the function returns a global/linear bin number.

```
Int_t bin = h->GetBin(binx,biny,binz);
```

This global bin is useful to access the bin information independently of the dimension.

2.5 Filling Histograms

A histogram is typically filled with statements like:

```
h1->Fill(x);
h1->Fill(x,w); //with weight
h2->Fill(x,y)
h2->Fill(x,y,w)
h3->Fill(x,y,z)
h3->Fill(x,y,z,w)
```

The `Fill` method compute the bin number corresponding to the given x, y or z argument and increment this bin by the given weight. The `Fill` method return the bin number for 1-D histograms or

global bin number for 2-D and 3-D histograms. If `TH1::Sumw2` has been called before filling, the sum of squares is also stored. One can also increment directly a bin number via `TH1::AddBinContent` or replace the existing content via `TH1::SetBinContent`. To access the bin content of a given bin, do:

```
Double_t binContent = h->GetBinContent(bin);
```

By default, the bin number is computed using the current axis ranges. If the automatic binning option has been set via: `h->SetBit(TH1::kCanRebin);` then, the `Fill` method will automatically extend the axis range to accommodate the new value specified in the `Fill` argument. The method used is to double the bin size until the new value fits in the range, merging bins two by two. This automatic binning options is extensively used by the `TTree::Draw` function when histogramming Tree variables with an unknown range. This automatic binning option is supported for 1-d, 2-D and 3-D histograms.

During filling, some statistics parameters are incremented to compute the mean value and root mean square with the maximum precision. In case of histograms of type `TH1C`, `TH1S`, `TH2C`, `TH2S`, `TH3C`, `TH3` a check is made that the bin contents do not exceed the maximum positive capacity (127 or 65535). Histograms of all types may have positive or/and negative bin contents.

2.6 Re-binning

At any time, an histogram can be re-binned via `TH1::Rebin`. This method returns a new histogram with the re-binned contents. If bin errors were stored, they are recomputed during the re-binning.

2.7 Associated Error

By default, for each bin, the sum of weights is computed at fill time. One can also call `TH1::Sumw2` to force the storage and computation of the sum of the square of weights per bin. If `Sumw2` has been called, the error per bin is computed as the sqrt(sum of squares of weights), otherwise the error is set equal to the sqrt(bin content). To return the error for a given bin number, do:

```
Double_t error = h->GetBinError(bin);
```

2.8 Associated function

One or more object (typically a `TF1`*) can be added to the list of functions (`fFunctions`) associated to each histogram. When `TF1::Fit` is invoked, the fitted function is added to this list. Given an histogram h, one can retrieve an associated function with:

```
TF1 *myfunc = h->GetFunction("myfunc");
```

2.9 Operations on histograms

Many types of operations are supported on histograms or between histograms:

- Addition of a histogram to the current histogram
- Additions of two histograms with coefficients and storage into the current histogram
- Multiplications and Divisions are supported in the same way as additions.
- The Add, Divide and Multiply functions also exist to add, divide or multiply a histogram by a function.

If a histogram has associated error bars (`TH1::Sumw2` has been called), the resulting error bars are also computed assuming independent histograms. In case of divisions, binomial errors are also supported.

2.10 Fitting histograms

Histograms (1-D, 2-D, 3-D, and Profiles) can be fitted with a user specified function via `TH1::Fit`. When a histogram is fitted, the resulting function with its parameters is added to the list of functions of this histogram. If the histogram is made persistent, the list of associated functions is also persistent. Given a pointer (see above) to an associated function `myfunc`, one can retrieve the function/fit parameters with calls such as:

```
Double_t chi2 = myfunc->GetChisquare();
Double_t par0 = myfunc->GetParameter(0); // value of 1st parameter
Double_t err0 = myfunc->GetParError(0);  // error on first parameter
```

2.11 Projections of Histograms

One can:

- make a 1-D projection of a 2-D histogram or Profile see functions `TH2::ProjectionX,Y`, `TH2::ProfileX,Y`, `TProfile::ProjectionX`
- make a 1-D, 2-D or profile out of a 3-D histogram see functions `TH3::ProjectionZ`, `TH3::Project3D`.

One can fit these projections via: `TH2::FitSlicesX,Y`, `TH3::FitSlicesZ`

2.12 Random numbers and histograms

`TH1::FillRandom` can be used to randomly fill an histogram using the contents of an existing `TF1` function or another `TH1` histogram (for all dimensions). For example the following two statements create and fill a histogram 10000 times with a default gaussian distribution of mean 0 and sigma 1:

```
TH1F h1("h1","histo from a gaussian",100,-3,3);
h1.FillRandom("gaus",10000);
```

`TH1::GetRandom` can be used to return a random number distributed according the contents of an histogram.

2.13 Making a copy of a histogram

Like for any other ROOT object derived from `TObject`, one can use the `Clone` method. This makes an identical copy of the original histogram including all associated errors and functions:

```
TH1F *hnew = (TH1F*)h->Clone();
hnew->SetName("hnew"); // recommended otherwise you
                       // get 2 histograms with  the same name.
```

2.14 Normalizing histograms

One can scale an histogram such that the bins integral is equal to the normalization parameter via

```
 TH1::Scale(Double_t norm);
```

2.15 Drawing histograms

Histograms are drawn via the `THistPainter` class. Each histogram has a pointer to its own pointer (to be usable in a multithreaded program). Many drawing options are supported. See `TH1::Draw` for more details.

The same histogram can be drawn with different options in different pads. When a histogram drawn in a pad is deleted, the histogram is automatically removed from the pad or pads where it was drawn. If a histogram is drawn in a pad, then filled again, the new status of the histogram will be automatically shown in the pad next time the pad is updated. One does not need to redraw the histogram.

To draw the current version of a histogram in a pad, one can use

```
h->DrawCopy();
```

This makes a clone (see `Clone` above) of the histogram. Once the clone is drawn, the original histogram may be modified or deleted without affecting the aspect of the clone. One can use `TH1::SetMaximu` and `TH1::SetMinimum` to force a particular value for the maximum or the minimum scale on the plot.

`TH1::UseCurrentStyle` can be used to change all histogram graphics attributes to correspond to the current selected style. This function must be called for each histogram. In case one reads and draws many histograms from a file, one can force the histograms to inherit automatically the current graphics style by calling before `gROOT->ForceStyle();`

The `TH1::Draw()` method has many draw options. You can combine them in a list separated by commas. For the most up to date list of the draw options please see:
http://root.cern.ch/root/html/TH1.html#TH1:Draw

2.16 Setting Drawing histogram contour levels (2-D hists only)

By default, contours are automatically generated at equidistant intervals. A default value of 20 levels is used. This can be modified via `TH1::SetContour` or `TH1::SetContourLevel`. The contours level info is used by the drawing options "`cont`", "`surf`", and "`lego`".

2.17 Setting histogram graphics attributes

The histogram classes inherit from the attribute classes: `TAttLine, TAttFill, TAttMarker` and `TAttText`. See the member functions of these classes for the list of option.

2.18 Giving the axis a title

```
h->GetXaxis()->SetTitle("X axis title");
h->GetYaxis()->SetTitle("Y axis title");
```

The histogram title and the axis titles can be any `TLatex` string. The titles are part of the persistent histogram.

2.20 Saving/Reading histograms to/from a Root file

The following statements create a ROOT file and store a histogram on the file. Because `TH1` derives from `TNamed`, the key identifier on the file is the histogram name:

```
TFile f("histos.root","new");
TH1F h1("hgaus","histo from a gaussian",100,-3,3);
h1.FillRandom("gaus",10000);
h1->Write();
```

To read this histogram in another ROOT session, do:

```
TFile f("histos.root");
TH1F *h = (TH1F*)f.Get("hgaus");
```

One can save all histograms in memory to the file by:

```
file->Write();
```

2.21 Miscellaneous histogram operations

`TH1::KolmogorovTest`	statistical test of compatibility in shape between two histograms.
`TH1::Smooth`	smoothes the bin contents of a 1-d histogram

`TH1::Integral`	returns the integral of bin contents in a given bin range
`TH1::GetMean(int axis)`	returns the mean value along axis
`TH1::GetRMS(int axis)`	returns the Root Mean Square along axis
`TH1::GetEntries`	returns the number of entries
`TH1::Reset()`	resets the bin contents and errors of an histogram.

3 COLLECTION CLASSES

Collections are a key feature of the ROOT framework. Many, if not most, of the applications you write will use collections. A collection is a group of related objects. You will find it easier to manage a large number of items as a collection. For example, a diagram editor might manage a collection of points and lines. A set of widgets for a graphical user interface can be placed in a collection. A geometrical model can be described by collections of shapes, materials and rotation matrices. Collections can themselves be placed in collections. Collections act as flexible alternatives to traditional data structures of computers science such as arrays, lists and trees.

3.1 General Characteristics

The ROOT collections are polymorphic containers that hold pointers to `TObjects`, so:

1) They can only hold objects that inherit from `TObject`
2) They return pointers to `TObjects` that have to be cast back to the correct subclass

Collections are dynamic and they can grow in size as required. They are descendants of `TObject` so can themselves be held in collections. It is possible to nest one type of collection inside another to any level to produce structures of arbitrary complexity.

Collections don't own the objects they hold for the very good reason. The same object could be a member of more than one collection. Object ownership is important when it comes to deleting objects; if nobody owns the object, it could end up as wasted memory (i.e. a memory leak) when no longer needed. If a collection is deleted, its objects are not. The user can force a collection to delete its objects, but that is the user's choice.

3.2 Types of Collections

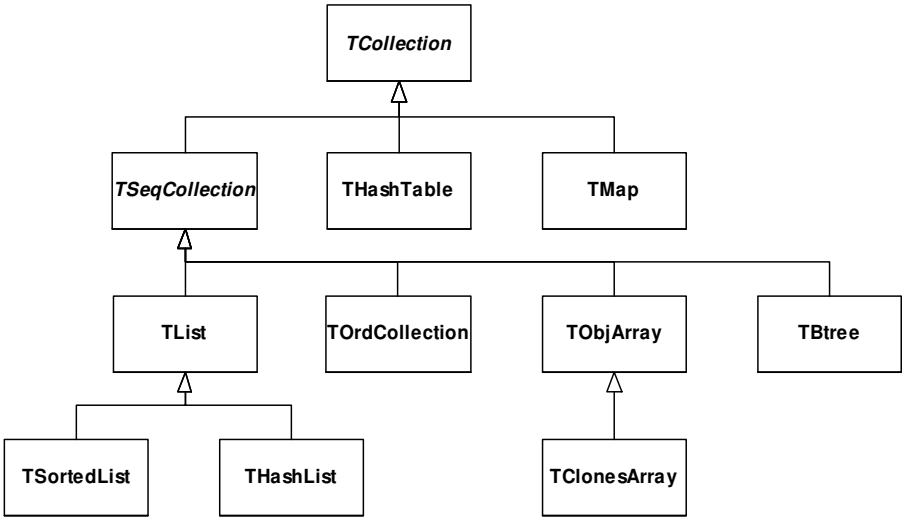

Figure 6: the Collection Classes

The ROOT system implements the following basic types of collections: unordered collections, ordered collections and sorted collections. This figure *s*hows the inheritance hierarchy for the primary collection classes. All primary collection classes derive from the abstract base class `TCollection`.

3.3 Ordered Collections (Sequences)

Sequences are collections that are externally ordered because they maintain internal elements according to the order in which they were added. The following sequences are available: `TList, THashList, TOrdCollection, TObjArray, and TClonesArray.`The `TOrdCollection`, `TObjArray` , as well as the `TClonesArray` can be sorted using their `Sort()` member function (if the stored items are sort-able). Ordered collections all derive from the abstract base class `TSeqCollection`.

3.4 Sorted Collection

Sorted collections are ordered by an internal (automatic) sorting mechanism. The following sorted collections are available: `TSortedList and TBtree.` The stored items must be sort-able.

3.5 Unordered Collection

Unordered collections don't maintain the order in which the elements were added, i.e. when you iterate over an unordered collection, you are not likely to retrieve elements in the same order they were added to the collection. The following unordered collections are available: `THashTable and Tmap.`

3.6 Iterators: Processing a Collection

The concept of processing all the members of a collection is generic, i.e. independent of any specific representation of a collection. To process each object in a collection one needs some type of cursor that is initialized and then steps over each member of the collection in turn. Collection objects could provide this service but there is a snag: as there is only one collection object per collection there would only be one cursor. Instead, to permit the use of as many cursors as required, they are made separate classes called iterators. For each collection class there is an associated iterator class that knows how to sequentially retrieve each member in turn. The relationship between a collection and its iterator is very close and may require the iterator to have full access to the collection (i.e. it is a friend). For example: `TList TListIter`, and `TMap TMapIter`. In general, iterators will be used via the `TIter` wrapper class.

3.7 A Collectable Class

The basic protocol the `TObject` class defines for collection elements are:

`IsEqual()`	Is used by the `FindObject()` collection method. By default `IsEqual()` compares the two object pointers.
`Compare()`	Returns –1, 0 or 1 depending if the object is smaller, equal or larger than the other object. By default a `TObject` has not a valid `Compare()` method.
`IsSortable()`	Returns true if the class is sortable (i.e. if it has a valid `Compare()` method). By default, a `TObject` is not sortable.
`Hash()`	Returns a hash value. This method needs to be implemented if an object has to be stored in a collection using a hashing technique, like `THashTable`, `THashList` and `TMap`. By default `Hash()` returns the address of the object. It is essential to choose a good hash function.

3.8 The `TIter` Generic Iterator

As stated above, the `TIterator` class is abstract; it is not possible to create `TIterator` objects. However, it should be possible to write generic code to process all members of a collection so there is a need for a generic iterator object. A `TIter` object acts as generic iterator. It provides the same `Next()` and `Reset()` methods as `TIterator` but it has no idea how to support them! It works as follows:

To create a `TIter` object its constructor must be passes an object that inherits from `TCollection`. The `TIter` constructor calls the `MakeIterator()` method of this collection to get the appropriate iterator object that inherits from `TIterator`.

The `Next()` and `Reset()` methods of `TIter` simply call the `Next()` and `Reset()` methods of the iterator object. So `TIter` simply acts as a wrapper for an object of a concrete class inheriting from TIterator. For example:

```
MyClass *myobject;
TList *mylist = GetPointerToList();
TIter myiter(mylist);
while ((myobject = (MyClass) myiter.Next())) {
   // process myobject
}
```

4 THE GUI CLASSES

Embedded in the ROOT system is an extensive set of GUI classes. The GUI classes provide a full OO-GUI framework as opposed to a simple wrapper around a GUI such as Motif. All GUI elements do their drawing via the `TGXW` low-level graphics abstract base class. Depending on the platform on which you run ROOT, the concrete graphics class (inheriting from `TGXW`) is either `TGX11` or `TGWin32`. All GUI widgets are created from "first principles", i.e., they only use routines like **`DrawLine`**, **`FillRectangle`**, **`CopyPixmap`**, etc., and therefore, the `TGX11` implementation only needs the X11 and Xpm libraries. The advantage of the abstract base class approach is that porting the GUI classes to a new, non X11/Win32, platform requires only the implementation of an appropriate version of `TGXW` (and of `TSystem` for the OS interface).

All GUI classes are fully scriptable and accessible via the interpreter. This allows for fast prototyping of widget layouts. They are based on the XClass'95 library written by David Barth and Hector Peraza. The widgets have the well-known Windows 95 look and feel. For more information on XClass'95, see ftp://mitac11.uia.ac.be/html-test/xclass.html.

5 INTEGRATING YOUR OWN CLASSES INTO ROOT

In this section, we'll give a step-by-step method for integrating your own classes into ROOT. Once integrated you can save instances of your class in a ROOT database, inspect objects at run-time, create and manipulate objects via the interpreter, generate HTML documentation, etc. A very simple class describing some person attributes is shown above. The Person implementation file Person.cxx is shown here.

```
#ifndef __PERSON_H
#define __PERSON_H
#include <TObject.h>

class Person : public TObject { // need to inherit from TObject

private:
   int age;      // age of person
   float height; // height of person

public:
   Person(int a = 0, float h = 0) : age(a), height(h) { }
   int get_age(void) const { return age; }
   float get_height(void) const { return height; }

   void set_age(int a) { age = a; }
   void set_height(float h) { height = h; }

   ClassDef(Person,1) // Person class
};
#endif
```

In this section, we'll give a step-by-step method for integrating your own classes into ROOT. Once integrated you can save instances of your class in a ROOT database, inspect objects at run-time, create and manipulate objects via the interpreter, generate HTML documentation, etc. A very simple class describing some person attributes is shown above. The Person implementation file Person.cxx is shown here.

```
#include "Person.h"
// ClassImp provides the implementation of some
// functions defined in the ClassDef script
ClassImp(Person)
```

The scripts **ClassDef** and **ClassImp** provide some member functions that allow a class to access its interpreter dictionary information. Inheritance from the ROOT basic object, `TObject`, provides the interface to the database and introspection services.

Now run the **rootcint** program to create a dictionary, including the special I/O streamer and introspection methods for class `Person`:

```
bash$ rootcint -f dict.cxx -c Person.h
```

Next compile and link the source of the class and the dictionary into a single shared library:

```
bash$ g++ -fPIC -I$ROOTSYS/include -c dict.cxx
bash$ g++ -fPIC -I$ROOTSYS/include -c Person.cxx
bash$ g++ -shared -o Person.so Person.o dict.o
```

Now start the ROOT interactive program and see how we can create and manipulate objects of class Person using the CINT C++ interpreter:

```
bash$ root
root [0] gSystem->Load("Person")
root [1] Person rdm(37, 181.0)
root [2] rdm.get_age()
(int)37
root [3] rdm.get_height()
(float)1.810000000000e+02
root [4] TFile db("test.root","new")
root [5] rdm.Write("rdm") // Write is inherited from theTObject class
root [6] db.ls()
TFile** test.root
 TFile* test.root
 KEY: Person rdm;1
root [7] .q
```

Here the key statement was the command to dynamically load the shared library containing the code of your class and the class dictionary.

In the next session, we access the **rdm** object we just stored on the database `test.root`.

```
bash$ root
root [0] gSystem->Load("Person")
root [1] TFile db("test.root")
root [2] rdm->get_age()
(int)37
root [3] rdm->Dump() // Dump is inherited from the TObject class
age              37          age of person
height           181         height of person
fUniqueID        0           object unique identifier
fBits            50331648    bit field status word
root [4] .class Person
[follows listing of full dictionary of class Person]
root [5] .q
```

```
 //----------- begin fill.C
{
   gSystem->Load("Person");
   TFile *f = new TFile("test.root","recreate");
   Person *t;
   for (int i = 0; i < 1000; i++) {
      char s[10];
      t = new Person(i, i+10000);
      sprintf(s, "t%d", i);
      t->Write(s);
      delete t;
   }
   f->Close();
}
```

A C++ script that creates and stores 1000 persons in a database is shown above. To execute this script, do the following:

```
bash$ root
root [0] .x fill.C
root [1] .q
```

This method of storing objects would be used only for several thousands of objects. The special Tree object containers should be used to store many millions of objects of the same class.

```
//----------- begin find.C
void find(int begin_age = 0, int end_age = 10)
{
   gSystem->Load("Person");
   TFile *f = new TFile("test.root");
   TIter next(f->GetListOfKeys());
   TKey *key;
   while (key = (TKey*)next()) {
      Person *t = (Person *) key->Read();
      if (t->get_age() >= begin_age && t->get_age() <= end_age) {
         printf("age = %d, height = %f\n", t->get_age(), t->get_height());
      }
      delete t;
   }
}
```

This listing is a C++ script that queries the database and prints all persons in a certain age bracket. To execute this script do the following:

```
bash$ root
root [0] .x find.C(77,80)
age = 77, height = 10077.000000
age = 78, height = 10078.000000
age = 79, height = 10079.000000
age = 80, height = 10080.000000
NULL
root [1] find(888,895)
age = 888, height = 10888.000000
age = 889, height = 10889.000000
age = 890, height = 10890.000000
age = 891, height = 10891.000000age = 892, height = 10892.000000
age = 893, height = 10893.000000
age = 894, height = 10894.000000
age = 895, height = 10895.000000
root [2] .q
```

With Person objects stored in a Tree, this kind of analysis can be done in a single command.

Finally, a small C++ script that prints all methods defined in class `Person` using the information stored in the dictionary is shown below:

```
//------------ begin method.C
{
   gSystem->Load("Person");
   TClass *c = gROOT->GetClass("Person");
   TList *lm = c->GetListOfMethods();
   TIter next(lm);
   TMethod *m;
   while (m = (TMethod *)next()) {
     printf("%s %s%s\n", m->GetReturnTypeName(), m->GetName(),
            m->GetSignature());
   }
}
```

To execute this script, type:

```
bash$ root
root [0] .x method.C
class Person Person(int a = 0, float h = 0)
int get_age()
float get_height()
void set_age(int a)
void set_height(float h)
const char* DeclFileName()
int DeclFileLine()
const char* ImplFileName()
int ImplFileLine()
Version_t Class_Version()
class TClass* Class()
void Dictionary()
class TClass* IsA()
void ShowMembers(class TMemberInspector& insp, char* parent)
void Streamer(class TBuffer& b)
class Person Person(class Person&)
void ~Person()
root [1] .q
```

The above examples prove the functionality that can be obtained when you integrate, with a few simple steps, your classes into the ROOT framework.

6 OBJECT PERSISTENCY

Object persistency, the ability to save and read objects, is fully supported by ROOT. The `TFile` and `TTree` classes make up the backbone of I/O. The `TFile` is a ROOT class encapsulating the behavior of a file containing ROOT objects, and the `TTree` is a data structure optimized to hold many same class objects.

6.1 ROOT Files

A ROOT file is like a UNIX file directory. It can contain directories and objects organized in unlimited number of levels. It also is stored in machine independent format (ASCII, IEEE floating point, Big Endian byte ordering). This is an example to explain the physical layout of a ROOT file. This example creates a ROOT file and 15 histograms, fills each histogram with 1000 entries from a gaussian distribution, and writes them to the file.

```
{
   TFile f("demo.root","recreate");
   char name[10], title[20];
   for (Int_t i = 0; i < 15; i++) {
      sprintf(name,"h%d",i);
      sprintf(title,"histo nr:%d",i);
      TH1F *h = new TH1F(name,title,100,-4,4);
      h->FillRandom("gaus",1000);
      h->Write();
   }
   f.Close();
}
```

The example begins with a call to the `TFile` constructor. `TFile` is the class describing the ROOT file. In the next section, when we discuss the logical file structure, we will cover `TFile` in detail. We can view the contents by creating a `TBrowser` object.

```
root [] TFile f("demo.root")
root [] TBrowser browser;
```

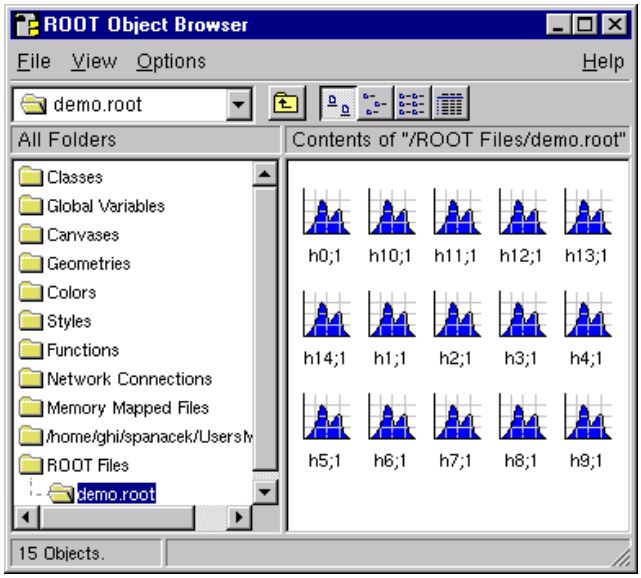

Figure 7: The Newly Created Histograms

In the browser, we can see the 15 histograms we created. Once we have the `TFile` object we can

call the `TFile::Map()` method to view the physical layout. . The first 64 bytes are taken by the file header. What follows are the 15 histograms in records of various length. Each record has a header (a `TKey`) with information about the object including its directory.

A ROOT file has a maximum size of two GB. It keeps track of the free space in terms of records and bytes. This count also includes the deleted records, which are available again.

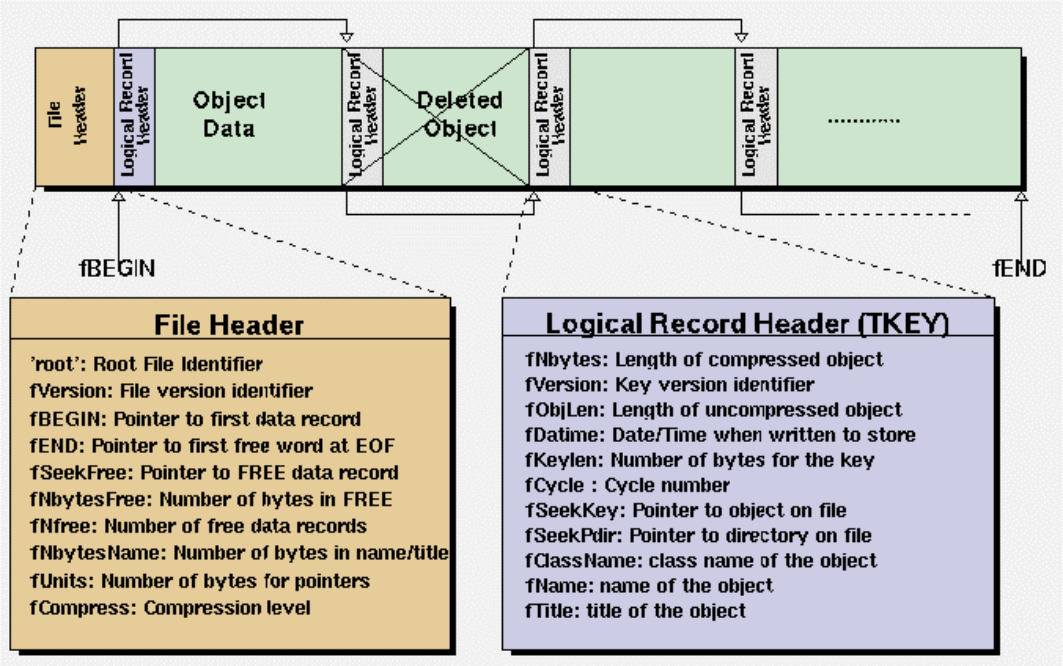

Figure 8: The ROOT file structure

6.2 File Recovery

A file may become corrupted or it may be impossible to write it to disk and close it properly. For example if the file is too large and exceeds the disk quota, or the job crashes or a batch job reaches its time limit before the file can be closed. In these cases, it is imminent to recover and retain as much information as possible. ROOT provides an intelligent and elegant file recovery mechanism using the redundant directory information in the record header.

If the file is not closed due to for example exceeded the time limit, and it is opened again, it is scanned and rebuilt according to the information in the record header. The recovery algorithm reads the file and creates the saved objects in memory according to the header information. It then rebuilds the directory and file structure. If the file is opened in write mode, the recovery makes the correction on disk when the file is closed; however if the file is opened in read mode, the correction can not be written to disk. You can also explicitly invoke the recovery procedure by calling the `TFile::Recover()` method. You must be aware of the 2GB size limit before you attempt a recovery. If the file has reached this limit, you cannot add more data. You can still recover the directory structure, but you cannot save what you just recovered to the file on disk.

6.3 Compression

The last parameter in the `TFile` constructor is the compression level. By default, objects are compressed before being written to a file. Data in the records can be compressed or uncompressed, but the record headers are never compressed. ROOT uses a compression algorithm based on the well-known gzip algorithm. This algorithm supports up to nine levels of compression, and the default ROOT uses is

one. The level of compression can be modified by calling the `TFile::SetCompressionLevel()` method. If the level is set to zero, no compression is done. Experience with this algorithm indicates a compression factor of 1.3 for raw data files and around two on most DST files is the optimum. The choice of one for the default is a compromise between the time it takes to read and write the object vs. the disk space savings. The time to uncompress an object is small compared to the compression time and is independent of the selected compression level. Note that the compression level may be changed at any time, but the new compression level will only apply to newly written objects. Consequently, a ROOT file may contain objects with different compression levels.

Compression	Bytes
0	13797
1	6290
5	6103
9	5912

The table shows four runs of the demo script that creates 15 histograms with different compression parameters.

6.4 The Logical ROOT File: TFile and TKey

We saw that the `TFile::Map()` method reads the file sequentially and prints information about each record while scanning the file. It is not feasible to only support sequential access and hence ROOT provides random or direct access, i.e. reading a specified object at a time. To do so, `TFile` keeps a list of `TKeys`, which is essentially an index to the objects in the file. The `TKey` class describes the record headers of objects in the file. For example, we can get the list of keys and print them. To find a specific object on the file we can use the `TFile::Get()` method.

```
root [] TFile f("demo.root")
root [] f.GetListOfKeys()->Print()
TKey Name = h0, Title = histo nr:0, Cycle = 1
…
TKey Name = h14, Title = histo nr:14, Cycle = 1
root [] TH1F *h9 = (TH1F*)f.Get("h9");
```

The `TFile::Get()` finds the `TKey` object with name "h9".

Since the keys are available in a `TList` of `TKeys`, we can iterate over the list of keys:

```
{
  TFile f("demo.root");
  TIter next(f.GetListOfKeys());
  TKey *key;
  while ((key=(TKey*)next())) {
    printf(
     "key: %s points to an object of class: %s at %d\n",
     key->GetName(),
     key->GetClassName(),key->GetSeekKey()
    );
  }
}
```

6.5 Viewing the Logical File Contents

`TFile` is a descendent of `TDirectory`, which means it behaves like a `TDirectory`. We can list the contents, print the name, and create subdirectories. In a ROOT session, you are always in a directory and the directory you are in is called the current directory and is stored in the global variable ***gDirectory***.

Here we open a file and list its contents:

```
root [] TFile f ("hsimple.root", "UPDATE")
root [] f.ls()
TFile** hsimple.root
TFile* hsimple.root
KEY: TH1F hpx;1 This is the px distribution
KEY: TH2F hpxpy;1 py vs px
KEY: TProfile hprof;1 Profile of pz versus px
KEY: TNtuple ntuple;1 Demo ntuple
```

It shows the two lines starting with `TFile` followed by four lines starting with the word "KEY". The four KEYs tell us that there are four objects on disk in this file. The syntax of the listing is:

```
KEY: <class> <variable>;<cycle number> <title>
```

For example, the first line in the list means there is an object in the file on disk, called `hpx`. It is of the class `TH1F` (one-dimensional histogram of floating-point numbers). The object's title is "This is the px distribution".

If the line starts with OBJ, the object is in memory. The <class> is the name of the root class (T-something). The <variable> is the name of the object. The cycle number along with the variable name uniquely identifies the object. The <title> is the string given in the constructor of the object as title.

6.6 Retrieving Objects from Disk

Multiple versions of an object with the same name, but unique cycle numbers can be in a ROOT file. ROOT automatically retrieves the one with the highest cycle number. To read an earlier version we have to explicitly specify the cycle number.

6.7 Subdirectories and Navigation

The `TDirectory` class lets you organize its contents into subdirectories, and `TFile` being a descendent of `TDirectory` inherits this ability.

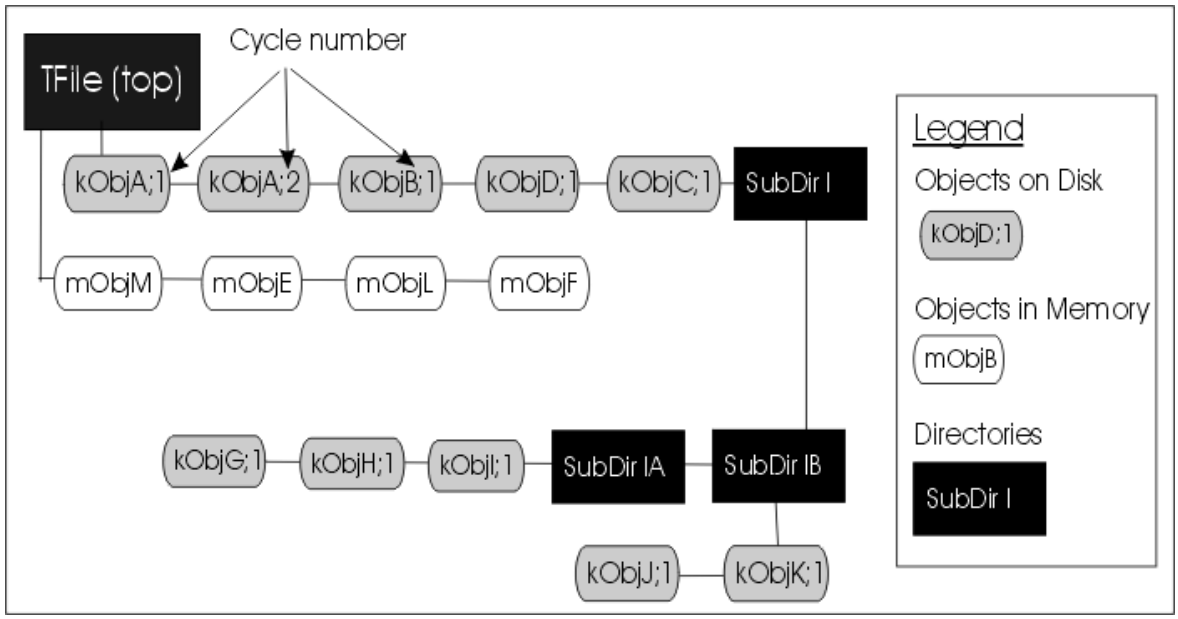

Figure 9: Objects in memory and objects on disk

This picture shows a `TFile` with five objects in the top directory (`kObjA;1, kObjA;2, kObjB;1, kObjC;1 and kObjD;1`). `ObjA` is on file twice with two different cycle numbers. It also shows four objects in memory (`mObjE, mObjeF, mObjM, mObjL`). It also shows several

subdirectories. Note that you could have the same named object, for example `kOBJA` , in a subdirectory and it would be unique because the entire path name is considered when identifying an object.

6.8 ROOT Trees

We explained how objects could be saved in ROOT files. In case you want to save large quantities of the same class objects, ROOT offers the `TTree` and `TNtuple` classes specifically for that purpose. The `TTree` class is optimized to reduce disk space and enhance access speed. A `TTree` can hold all kinds of data, such as objects or arrays in addition to all the simple types. A `TNtuple` is a `TTree` that is limited to only hold floating-point numbers.

We can use an example to illustrate the difference in saving individual objects vs. filling and saving the tree. Let's assume we have one million events and we write each one to a file, not using a tree. The `TFile`, being unaware that an event is always an event and the header information is always the same, will contain one million copies of the header. This header is about 60 bytes long, and contains information about the class, such as its name and title. For one million events, we would have 60 million bytes of redundant information. For this reason, ROOT gives us the `TTree` class. A `TTree` is smart enough not to duplicate the object header, and is able to reduce the header information to about four bytes per object, saving 56 million bytes in our case.

When using a `TTree`, we fill its branch buffers with data and the buffer is written to file when it is full. Branches and buffers are explained below. It is important to realize that not each object is written out individually, but rather collected and written a bunch at a time. In our example, we would fill the `TTree` with the million events and save the tree incrementally as we fill it. The `TTree` is also used to optimize the data access. A tree uses a hierarchy of branches, and each branch can be read independently from any other branch.

Assume that our Event class has 100 data members and `Px` and `Py` are two of them. We would like to compute $Px^2 + Py^2$ for every event and histogram the result. If we had saved the million events without a `TTree` we would have to: 1) read each event in its entirety (100 data members) into memory, 2) extract the `Px` and `Py` from the event, 3) compute the sum of the squares, and 4) fill a histogram. We would have to do that a million times! This is very time consuming, and we really do not need to read the entire event, every time. All we need are two little data members (`Px` and `Py`).

If we used a tree with one branch containing `Px` and another branch containing `Py`, we can read all values of `Px` and `Py` by only reading the `Px` and `Py` branches. This is much quicker since the other 98 branches (one per data member) are never read. This makes the use of the `TTree` very attractive.

6.9 Branches

A `TTree` is organized into a hierarchy of branches. These typically hold related variables or 'leaves'. By now, you probably guessed that the class for a branch is called `TBranch`. The organization of branches allows the designer to optimize the data for the anticipated use.

If two variables are independent, and the designer knows the variables will not be used together, they should be placed on separate branches. If, however, the variables are related, such as the coordinates of a point, it is most efficient to create one branch with both coordinates on it. A variable on a `TBranch` is called a leaf (yes - `TLeaf`). Another point to keep in mind when designing trees is the branches of the same `TTree` can be written to separate files. To add a `TBranch` to a `TTree` we call the `TTree::Branch()` method. The branch type differs by what is stored in them. A branch can hold an entire object, a list of simple variables, or an array of objects. Each branch has a branch buffer that is used to collect the values of the branch variable (leaf). Once the buffer is full, it is written to the file. Because branches are written individually, it saves writing empty or half filled branches.

6.10 Autosave

Autosave gives the option to save all branch buffers every `n` bytes. We recommend using Autosave for large acquisitions. If the acquisition fails to complete, you can recover the file and all the contents since the last Autosave. To set the number of bytes between Autosave you can use the

`TTree::SetAutosave()` method. You can also call `TTree::Autosave` in the acquisition loop every `n` entries.

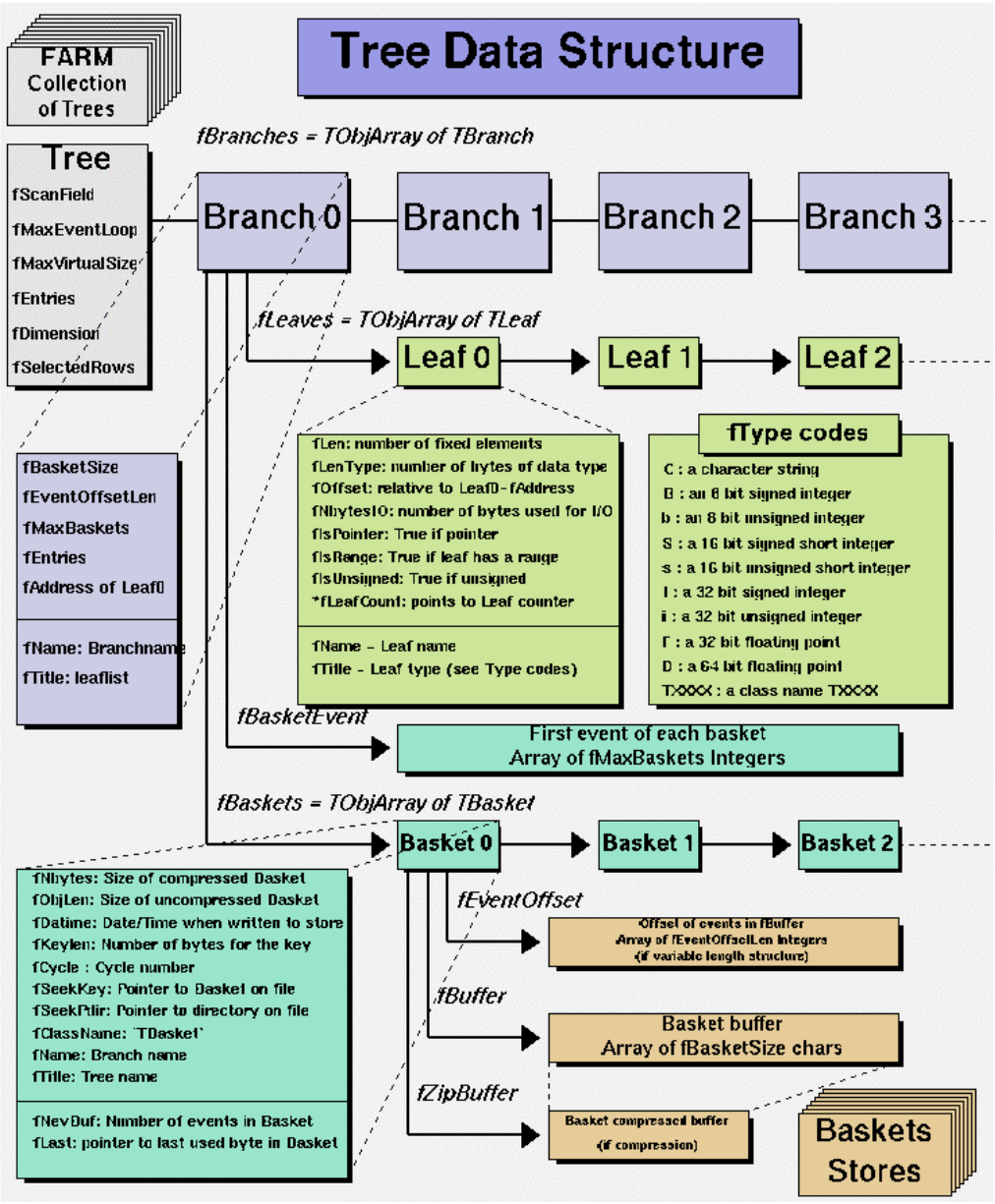

Figure 10 The Tree structure

6.11 Using Trees in Analysis

ROOT trees are not just optimized for access speed and data storage, but they also have a good set of built in analysis methods. There are several ways to analyze data stored in a ROOT Tree

Using `TTree::Draw`:

This is very convenient and efficient for small tasks. A `TTree::Draw` method produces one histogram at the time. The histogram is automatically generated. The selection expression may be specified in the command line.

Using the `TTreeViewer`:

This is a graphical interface to `TTree::Draw` with the same functionality.

Using the code generated by `TTree::MakeClass`:

In this case, the user creates an instance of the analysis class. He has the control over the event loop and he can generate an unlimited number of histograms.

Using the code generated by `TTree::MakeSelector`:

Like for the code generated by `TTree::MakeClass`, the user can do complex analysis. However, he cannot control the event loop. The event loop is controlled by `TTree::Process` called by the user. This solution is illustrated in the example analysis section at the end of this paper. The advantage of this method is that it can be run in a parallel environment using PROOF (the Parallel Root Facility).

6.12 The Tree Viewer

To bring up a tree viewer call the `TTree::StartViewer()` method. This pictureshows the Tree Viewer with several floating-point numbers leafs.

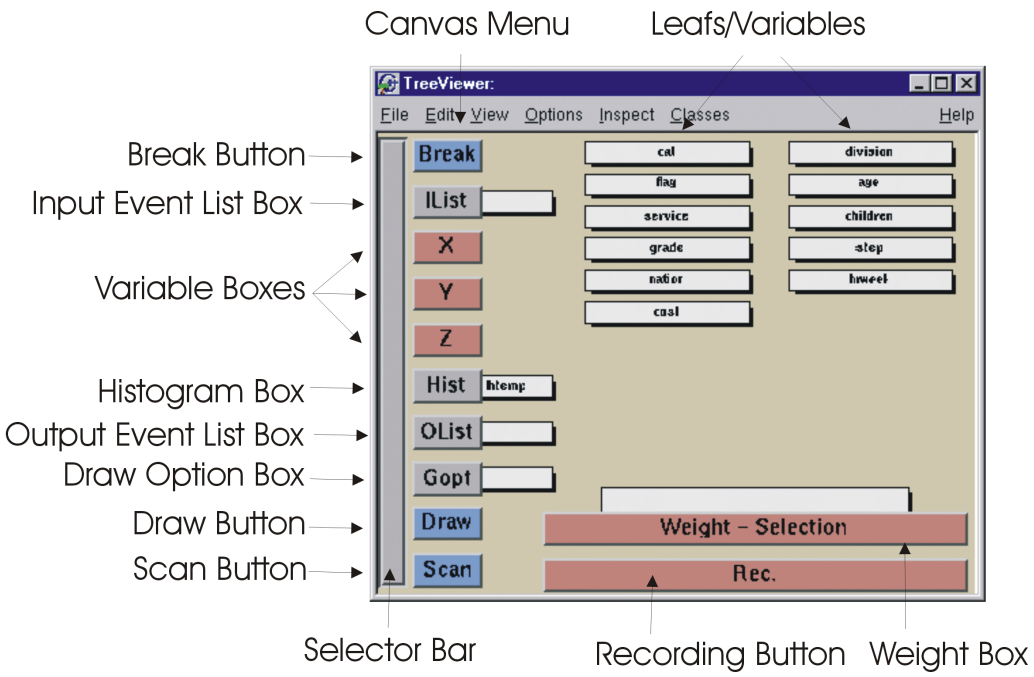

Figure 11: The Tree Viewer

The tree viewer allows you to quickly histogram any variable in the tree by double clicking on the leaf. With it, one can draw Lego plots, add cuts, select a list of events, and much more.

6.13 Simple Analysis using TTree::Draw

To show the different `Draw` options, we create a canvas with four sub-pads. We will use one sub-pad for each `Draw` command.

```
root [] TCanvas *myCanvas = new TCanvas("c","C", 0,0,600,400)
root [] myCanvas->Divide(2,2)
root [] myCanvas->cd(1)
root [] MyTree->Draw("fNtrack");
```

As you can see this signature of `Draw` has only one parameter. It is a string containing the leaf name. We activate the second pad and use this version of Draw:

```
root [] myCanvas->cd(2)
root [] MyTree->Draw("sqrt(fNtrack):fNtrack");
```

This signature still only has one parameter, but it now has two dimensions separated by a colon (“x:y”). The item to be plotted can be an expression not just a simple variable. In general, this parameter is a string that contains up to three expressions, one for each dimension, separated by a colon (“e1:e2:e3”).

Change the active pad to 3, and add a selection to the list of parameters of the draw command.

```
root []myCanvas->cd(3)
root []MyTree->Draw("sqrt(fNtrack):fNtrack","fTemperature > 20.8");
```

This will draw the fNtrack for the entries with a temperature above 20.8 degrees. In the selection parameter, you can use any C++ operator, plus some functions defined in TFormula. The next parameter is the draw option for the histogram:

```
root []myCanvas->cd(4)
root []MyTree->Draw("sqrt(fNtrack):fNtrack", "fTemperature > 20.8","surf2”);
```

There are many draw options. You can combine them in a list separated by commas. For the most up to date list of the draw options please see: http://root.cern.ch/root/html/TH1.html#TH1:Draw

There are two more optional parameters to the Draw method: one is the number of entries and the second one is the entry to start with.

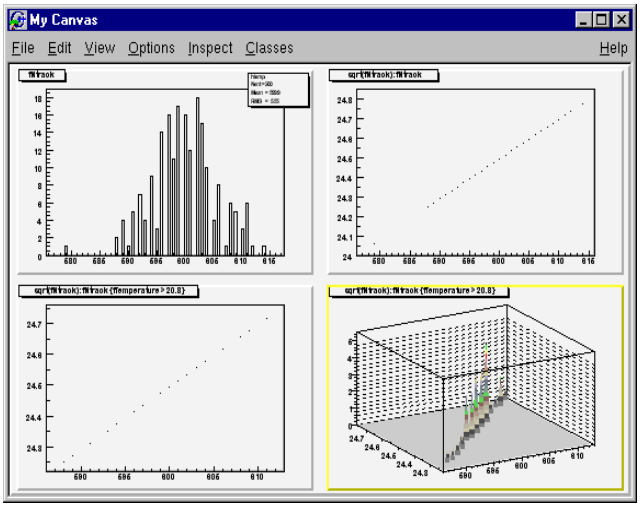

Figure 12: Draw Options

6.14 Creating an Event List

The `TTree::Draw` method can also be used to build a list of the selected entries. When the first argument is preceded by `">>"` ROOT knows that this command is not intended to draw anything, but to save the entries in a list with the name given by the first argument. The resulting list is a `TEventList`, and is added to the objects in the current directory. For example, to create a `TEventList` of all entries with more than 600 tracks:

```
root [] TFile *f = new TFile("AFile.root")
root [] T->Draw(">> myList", " fNtrack > 600")
```

This list contains the entry number of all entries with more than 600 tracks. To see the entry numbers use the `TEventList::Print("all")` method.

When using the ">>" whatever was in the TEventList is overwritten. The TEventList can be grown by using the ">>+" syntax. For example to add the entries, with exactly 600 tracks:

```
root [] T->Draw(">>+ myList", " fNtrack == 600")
```

If the Draw command generates duplicate entries, they are not added to the list.

6.15 Using an Event List

The TEventList can be used to limit the TTree to the events in the list. The SetEventList method tells the tree to use the event list and hence limits all subsequent TTree methods to the entries in the list. In this example, we create a list with all entries with more than 600 tracks and then set it so the Tree will use this list. To reset the TTree to use all events use SetEventList(0).

In the code snipped below, the entries with over 600 tracks are saved in a TEventList called myList. We get the list from the current directory and assign it to the variable list. Then we instruct the tree T to use the new list and draw it again.

```
root [] T->Draw(">>myList", " fNtrack >600")
root [] TEventList *list = (TEventList*)gDirectory->Get("myList")
root [] T->SetEventList(list)
root [] T->Draw("fNtrack ")
```

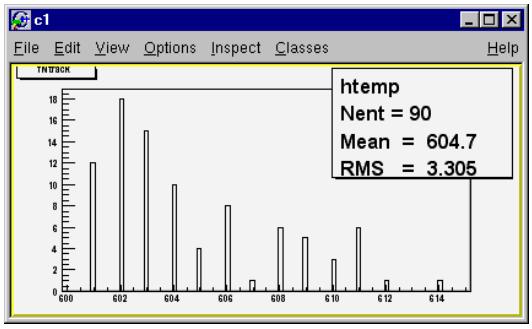

Figure 13: Histogram with the events from the Event list

6.16 Creating a Histogram

The TTree::Draw method can also be used to fill a specific histogram. The syntax is:

```
root [] TFile *f = new TFile("AFile.root")
root [] T->Draw("fNtrack >> myHisto")
root [] myHisto->Print()
TH1.Print Name= myHisto, Total sum= 200
```

As we can see, this created a TH1, called myHisto. If you want to append more entries to the histogram, you can use this syntax:

```
root [] T->Draw("fNtrack >>+ myHisto")
```

If you want to fill a histogram, but not draw it you can use the TTree::Project() method.

```
root [] T->Project("quietHisto","fNtrack")
```

6.17 Tree Information

Once we have drawn a tree, we can get information about the tree. These are the methods used to get information from a drawn tree:

- GetSelectedRows: Returns the number of entries accepted by the selection expression.

In case where no selection was specified, it returns the number of entries processed.

- GetV1: Returns a pointer to the double array of the first variable.
- GetV2: Returns a pointer to the double array of second variable
- GetV3: Returns a pointer to the double array of third variable.
- GetW: Returns a pointer to the double array of weights where the weight equals the result of the selection expression.

For a complete description and examples of these methods see:

http://root.cern.ch/root/html/TTree.html#TTree:Draw

6.18 More Complex Analysis using TTree::MakeClass

The Draw method is convenient and easy to use, however it falls short if you need to do some programming with the variable.

For example, for plotting the masses of all oppositely changed pairs of tracks, you would need to write a program that loops over all events, finds all pairs of tracks, and calculates the required quantities. We have shown how to retrieve the data arrays from the branches of the tree in the previous section, and you could write that program from scratch. Since this is a very common task, ROOT provides two utilities that generate a skeleton class designed to loop over the entries of the tree. The TTree::MakeClass and the TTree::MakeSelector methods. TTree::MakeSelector is explained and used in the next section with an example analysis.

6.19 Creating a Class with MakeClass

We load the shared library libEvent.so to make the class definitions available, and open the ROOT file. In the file, we see the TTree, and we use the TTree::MakeClass method on it. MakeClass takes one parameter, a string containing the name of the class to be made. In the command below, the name of our class will be "MyClass".

```
root [] .L libEvent.so
root [] TFile *f = new TFile ("Event.root");
root [] f->ls();
TFile**         Event.root      TTree benchmark ROOT file
 TFile*         Event.root      TTree benchmark ROOT file
  KEY: TH1F     htime;1 Real-Time to write versus time
  KEY: TTree    T;1     An example of a ROOT tree
  KEY: TH1F     hstat;1 Event Histogram
root [] T->MakeClass("MyClass")
Files: MyClass.h and MyClass.C generated from Tree: T
(Int_t)0
```

The call to MakeClass created two files. MyClass.h contains the class definition and several methods, and MyClass.C contains MyClass::Loop. The .C file is kept as short as possible, and contains only code that is intended for customization. The .h file contains all the other methods. Here is a description of each method.

- MyClass(TTree *tree=0): This constructor has an optional tree for a parameter. If you pass a tree, MyClass will use it rather than the tree from witch it was created.
- Void Init(TTree *tree): Init is called by the constructor to initialize the tree for reading. It associates each branch with the corresponding leaf data member.
- ~MyClass(): This is the destructor, nothing special.
- Int_t GetEntry(Int_t entry): This loads the class with the entry specified. Once you have executed GetEntry, the leaf data members in MyClass are set to the values of the entry. For

example, `GetEntry(12)` loads the 13th event into the event data member of MyClass (note that the first entry is 0). `GetEntry` returns the number of bytes read.

- `Int_t LoadTree(Int_t entry)` and `void Notify()`:
  These two methods are related to chains. `LoadTree` will load the tree containing the specified entry from a chain of trees. `Notify` is called by `LoadTree` to adjust the branch addresses.
- `void Loop()`: This is the skeleton method that loops through each entry of the tree. This is interesting to us, because we will need to customize it for our analysis.

6.20 Customizing the Loop

Here we see the implementation of `MyClass::Loop()`. `MyClass::Loop` consists of a for-loop calling `GetEntry` for each entry. In the skeleton, the numbers of bytes are added up, but it does nothing else. If we were to execute it now, there would be no output.

```
void MyClass::Loop()
{
   Int_t nentries = Int_t(fTree->GetEntries());
   Int_t nbytes = 0, nb = 0;
   for (Int_t i=0; i<nentries;i++) {
      if (LoadTree(i) < 0) break;
      nb = fTree->GetEntry(i);   nbytes += nb;
   }
}
```

For analysis, we modify this method. For example, here we select the first 100 tracks of each event and fill a histogram with it.

```
void MyClass::Loop()
{
 Track *track  = 0; Int_t n_Tracks = 0;
  TH1F *myHisto    = new TH1F("myHisto","fPx", 100, -5,5);
  TH1F *smallHisto = new TH1F("small","fPx", 100, -5,5);
  for (Int_t i=0; i<nentries;i++) {
     if (LoadTree(i) < 0) break;
     GetEntry(i); n_Tracks = event->GetNtrack();
     for (Int_t j = 0; j < n_Tracks; j++){
       track = (Track*) event->GetTracks()->At(j);
       myHisto->Fill(track->GetPx());
       if (j < 100){ smallHisto->Fill(track->GetPx()); }
     }
  }
  myHisto->Draw();
  smallHisto->Draw("Same");
}
```

Loading MyClass

To run the analysis the new class (`MyClass`) needs to be loaded and instantiated.

```
root [] .L libEvent.so
root [] .L MyClass.C
root [] MyClass m
```

Now we can call the methods of `MyClass`. For example, we can get a specific entry. In the code snipped below, we get entry 0, and print the number of tracks (594). Then we get entry 1 and print the number of tracks (597).

```
root [] m.GetEntry(0)
(int)1
root [] m.event->GetNtrack()
(Int_t)594
root [] m.GetEntry(1)
(int)48045
root [] m.event->GetNtrack()
(Int_t)597
root [] m.Loop()
```

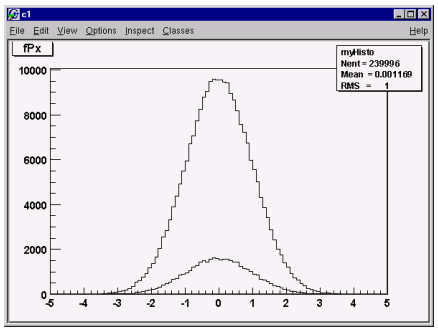

Figure 14: Results of the event loop

7 PROOF: ROOT'S PARALLEL PROCESSING FACILITY

Building on the experience gained from the implementation and operation of the PIAF system we have developed the parallel ROOT facility, PROOF. The main problems with PIAF were because its proper parallel operation depended on a cluster of homogenous equally performing and equally loaded machines. Due to PIAF's simplistic portioning of a job in N equal parts, where N is the number of processors, the overall performance was governed by the slowest node. The running of a PIAF cluster was an expensive operation since it required a cluster dedicated solely to PIAF. The cluster could not be used for other types of jobs without destroying the PIAF performance.

In the implementation of PROOF, we made the slave servers the active components that ask the master server for new work whenever they are ready. In the scheme the parallel processing performance is a function of the duration of each small job, packet, and the networking bandwidth and latency. Since the bandwidth and latency of a networked cluster are fixed the main tunable parameter in this scheme is the packet size. If the packet size is too small the parallelism will be destroyed by the communication overhead caused by the many packets sent over the network between the master and the slave servers. If the packet size is too large, the effect of the difference in performance of each node is not evened out sufficiently.

Another very important factor is the location of the data. In most cases, we want to analyze a large number of data files, which are distributed over the different nodes of the cluster. To group these files together we use a chain. A chain provides a single logical view of the many physical files. To optimize performance by preventing huge amounts of data being transferred over the network via NFS or any other means when analyzing a chain, we make sure that each slave server is assigned a packet, which is local to the node. Only when a slave has processed all its local data will it get packets assigned that cause remote access. A packet is a simple data structure of two numbers: begin event and number of events. The master server generates a packet when asked for by a slave server, taking into account t the time it took to process the previous packet and which files in the chain are local to the lave server. The master keeps a list of all generated packets per slave, so in case a slave dies during processing, all its packets can be reprocessed by the left over slaves.

8 USEFUL LINKS

The ROOT website is at: http://root.cern.ch
Various papers and talks on ROOT: http://root.cern.ch/root/Publications.html
Documentation on each ROOT class: http://root.cern.ch/root/html/ClassIndex.html
Documentation on IO: http://root.cern.ch/root/InputOutput.html

9 SUPPORTED PLATFORMS AND COMPILERS

ROOT is available on these platform/compiler combinations:

- Intel x86 Linux (g++, egcs and KAI/KCC)
- Intel Itanium Linux (g++)
- HP HP-UX 10.x (HP CC and aCC, egcs1.2 C++ compilers)
- IBM AIX 4.1 (xlc compiler and egcs1.2)
- Sun Solaris for SPARC (SUN C++ compiler and egcs)
- Sun Solaris for x86 (SUN C++ compiler)
- Compaq Alpha OSF1 (egcs1.2 and DEC/CXX)
- Compaq Alpha Linux (egcs1.2)
- SGI Irix (g++ , KAI/KCC and SGI C++ compiler)
- Windows NT and Windows95 (Visual C++ compiler)
- Mac MkLinux and Linux PPC (g++)
- Hitachi HI-UX (egcs)
- LynxOS
- MacOS (CodeWarrior, no graphics)