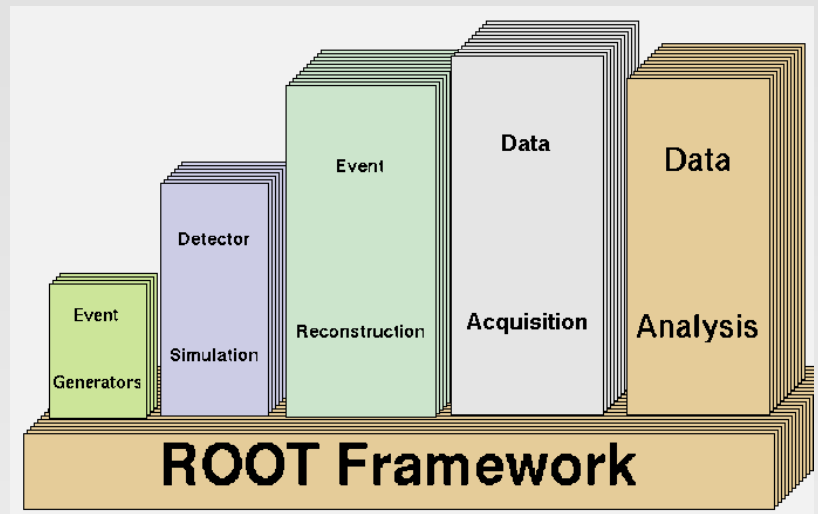
The ROOT framework

done by other software run in physics:

- Event generation and detector simulation
- Data Acquisition
- Data Storage
- Data Analysis



- It permits an easy management of large-scale experiments with many subsystems involved.

Utilities / Services of ROOT

  - Using low-level C++ calls ROOT can retrieve data stored on memory of sent by the DAQ.
- **Save Data**
  - ROOT provides a data structure (called tree) that is extremely powerful for fast access of huge amounts of data much faster than accessing a normal file
- **Access Data**
  - ROOT trees spread over several files can be chained and accessed as a unique object, allowing for loops over huge amounts of data.

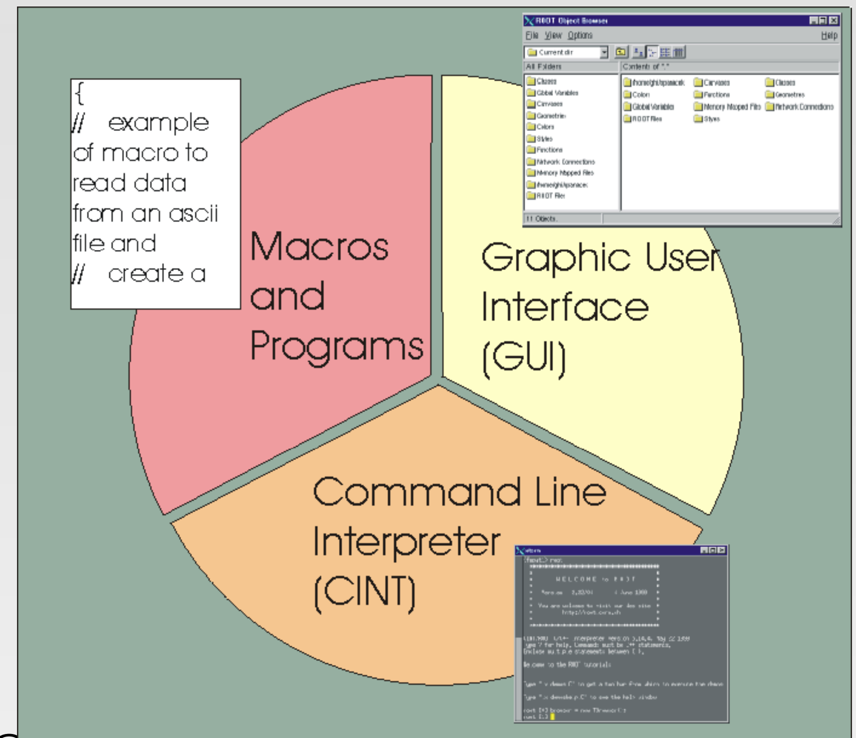
Utilities / Services of ROOT

- ROOT has powerful mathematical and statistical tools are provided to operate on your data. Fitting and random number generation are its strongpoints.

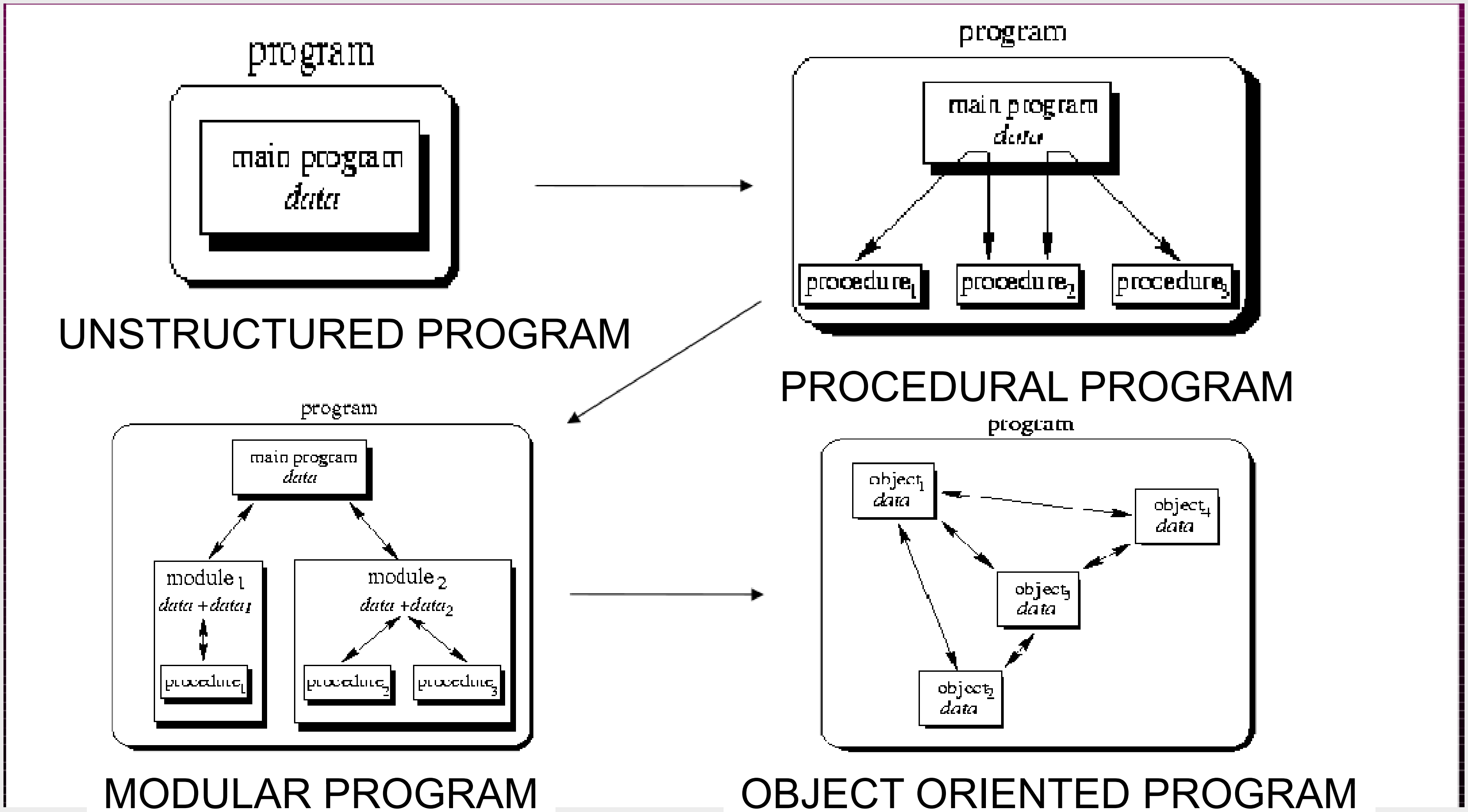
- **Show Results**
  - Publication-quality histograms, scatter plots, fitting functions, etc. may be shown and adjusted real-time.

3 ways of using ROOT

- ROOT can be coded interactively:
  - Just like a command-line interpreter, e.g. Bash. This mode uses the CINT syntax.
  - You can write macros that are interpreted by CINT
- ROOT programs can be compiled in for better performance.



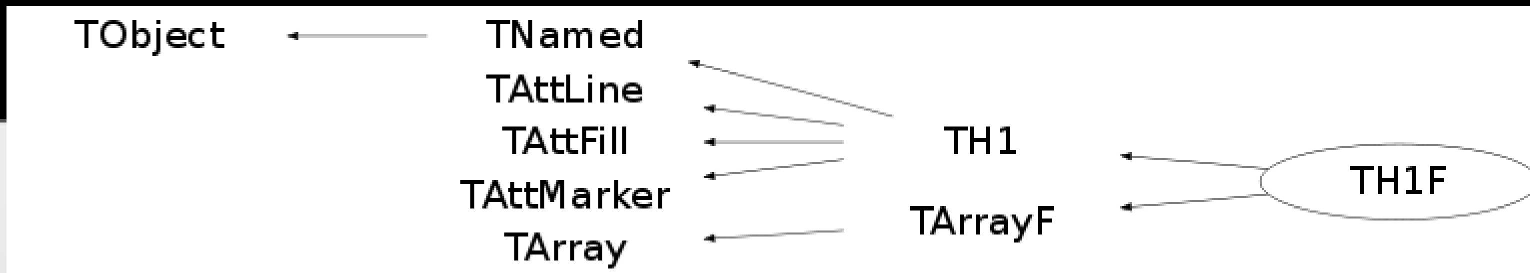
Architecture



Terminology in OO program

- Example: **TH1F** is the class defining a ROOT 1-D histogram with floating-point (32 bits) values.
- **Object:** the instance of a class
  - To define an histogram, one declares it as TH1F hist;
- **Method:** a function of the class
  - Example: hist.Draw() calls the funcion that draws.
- **Member:** data structure stored in the class
- **Inheritance:** the class ”is the daughter” of another, and inherits some of its mother's members

Class inheritances for TH1F



http://root.cern.ch/root/html/ClassIndex.html

http://root.cern.ch/drupal/content/architectural-overview

Resources for self-help

simple arrays to spectrum analyzers. Knowing all of them by heart is not easy.

- Extensive additional documentation is provided in the website:
  - User's Guide
  - Tutorials
  - HowTo's
  - FAQ's

First steps into ROOT

- [illegible]
  - $ROOTSYS is set and so are $PATH and $LD_LIBRARY_PATH
- Typing ”root” at the shell will get you to:
  - **root [0]**
- Welcome to the CINT, the ROOT C++ interpreter!
  - It is not a compiler, executes commands right away
  - It has auto-completion features and other amenities
  - It is less stable than you'd like it to be :(

Hello World 1 / 4

- [illegible]
  - **root [0] cout << "Hello World!" << endl;**
    Hello World!
- cout and << are the commands in the standard C++ namespace for printing out to std output.
- Hello world example 1.1 – char[] variable
  - **root [1] char hwrld_c[12] = "Hello World!"**
  - **root [2] cout << hwrld_c << endl;**
    Hello World!
- Here we declared an array of 12 chars, initializing it to "Hello World!", and printed it.

Hello World 2 / 4

- root [3] TString hwrld_s = ”Hello World!”;
- root [4] cout << hwrld_s << endl;
  Hello World!
- root [5] Int_t len = hwrld_s.Length();
- root [6] cout << len << endl;
  12

- Here we used the built-in ROOT TString class.
  As you can see the **instance** of the TString is called hwrld_s , and one of its **methods** is Length(), which returns an integer.

Hello World 3 / 4

- root [10] TPaveLabel hwrld_p(→

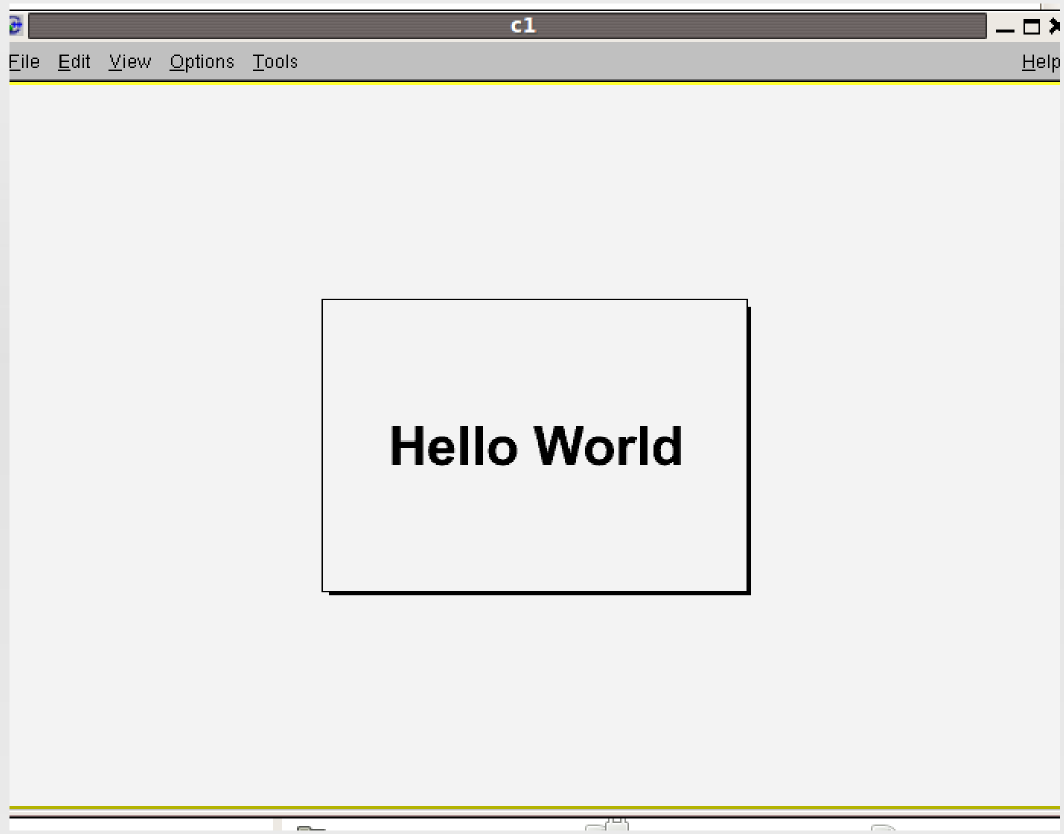
TPaveLabel TPaveLabel()

TPaveLabel TPaveLabel(Double_t x1, Double_t y1, Double_t x2, Double_t y2, const char* label, Option_t* option = "br")

TPaveLabel TPaveLabel(const TPaveLabel& pavelabel)

- root [11] TPaveLabel hwrld_p(0.3,0.3,0.7,0.7,"Hello World!","brNDC")
- root [12] hwrld_p.Draw();

Hello World 3 / 4



- In this last example, we first declared an instance to the class TPaveLabel
- With the command **(...)** we constructed it
- As before, the methods of this instance are called with ”.”
- Since this is a graphical class, CINT has automatically spawned a **TCanvas** where to draw.
- You can play with your mouse over it now and change it.

Hello World 4 / 4 : the macro

- myMacro.cxx

```
void myMacro(){
   cout << "Hello World!" << endl;
  }
```

- Then, you can run it from inside CINT:
  - root [14].x myMacro.cxx
- Or directly from the shell:
  - $ root -q -b myMacro.cxx
- Will give you the ability of reviewing your work as you go. -q and -b mean ”quiet” and ”batch”.

Hello World 4 / 4 : the macro

- myMacro2.cxx

```
Int_t myMacro2(Int_t k=0){
   cout << "The input is " << k << endl;
   return k;
   }
Double_t anotherFunc(Int_t j=0){
   Double_t pp = 2*TMath::ACos(-1);
   Double_t x = 1.5*j + pp*myMacro2(j);
   return x;
   }
```

- To be able to use anotherFunc, **load** the macro,
  - root [] .L myMacro.cxx

Variable Types in ROOT

- Basic types: capitalised and have suffix “_t”:
  int → Int_t   float → Float_t   double → Double_t
- Names of classes start with “T”:
  TH1F, TF1, TString, TDirectory, TFile, TTree...

- Some ROOT types (classes):
  - **TH1F** - Histogram, containing Float_t objects (floats)
  - **TString** – String container
  - **TF1** – 1-dimensional function, TF2, ...
  - **TTree** – can store per-event info
- see http://root.cern.ch/root/html/ListOfTypes.html

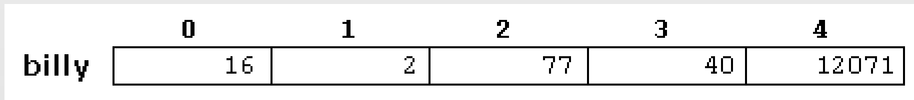
C++ operations within ROOT

- ```
  root [0] Int_t a;
  root [1] a = 5.1;
  root [2] cout << "a = " << a << endl;
  a = 5
  root [3] Double_t b;
  root [4] b = 5.1;
  root [5] cout << "b = " << b << endl;
  b = 5.1
  ```

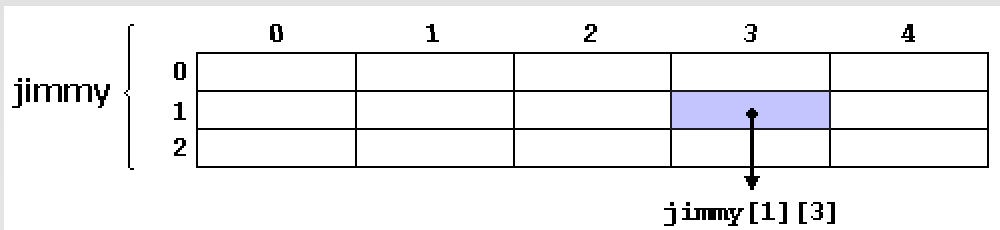
- Loops and controls: e.g. for loop with if/else
  - ```
    for (Int_t i=1; i < 10 ; i++ ) {
    if (i%2 == 0 ) cout << i << ” is even” << endl;
    else cout << i << ” is odd” << endl;
    }
    ```

C++ operations within ROOT

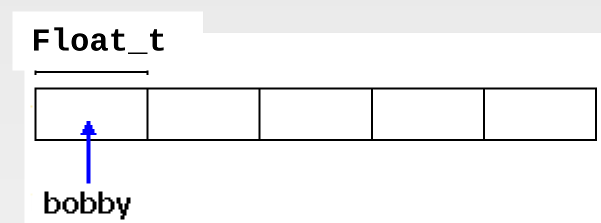
- root [] Int_t billy[5] = { 16, 2, 77, 40, 12071 };



- root [] Int_t jimmy[3][5];



- Pointers and dynamic memory:
  - root [] Float_t *bobby; // this is a pointer
    root [] Int_t narr = 100;
    root [] bobby = new Float_t [narr];
    bobby
    (Float_t*)0x8f1a4c8

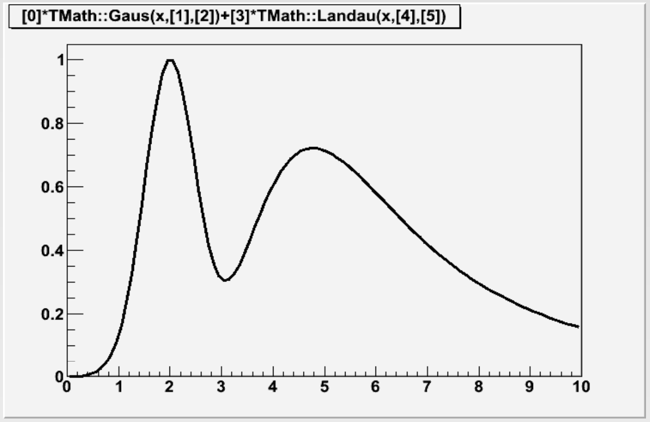


C++ resources

- ”course”:
  www.cplusplus.com
- C++ for ROOT users, from FNAL:
  http://www-root.fnal.gov/root/CPlusPlus/
- Standard template library (advanced stuff):
  http://www.sgi.com/tech/stl/

Functions: TMath and TF1

- TMath class. You can call them directly:
  - root [] TMath::TanH(1)
    (Double_t)7.61594155955764851e-01
- You can also define your own functions, using TF1:
  - root [] TF1 *f(
    = new TF1("test_f",
    "[0]*TMath::Gaus(x,[1],[2]) +
    [3]*TMath::Landau(x,[4],[5])",
    0.0,10.0);



  - root [] f->SetParameters(1.0,2,0.5,4,5,1);
    root [] f->Draw()

More math niceties

- TRandom class and its daughters:
  ```
  root [] TRandom3 rnd;
  root [] rnd.SetSeed(123456);
  root [] rnd.Poisson(3.4)
  (Int_t)2
  root [] f->GetRandom(0.0,10.0)
  (Double_t)2.08103799934920897e+00
  ```
- Physics vectors used to represent spacetime vectors and their transformations:
  ```
  root [] TVector3 r(1,0,0);
  root [] r.Rotate(TMath::Pi()/6.0,TVector3(0,1,0));
  root [] cout << r.Z() << endl;
  -0.5
  root [] TLorentzVector rl(r,1.0)
  ```

Histograms: TH1

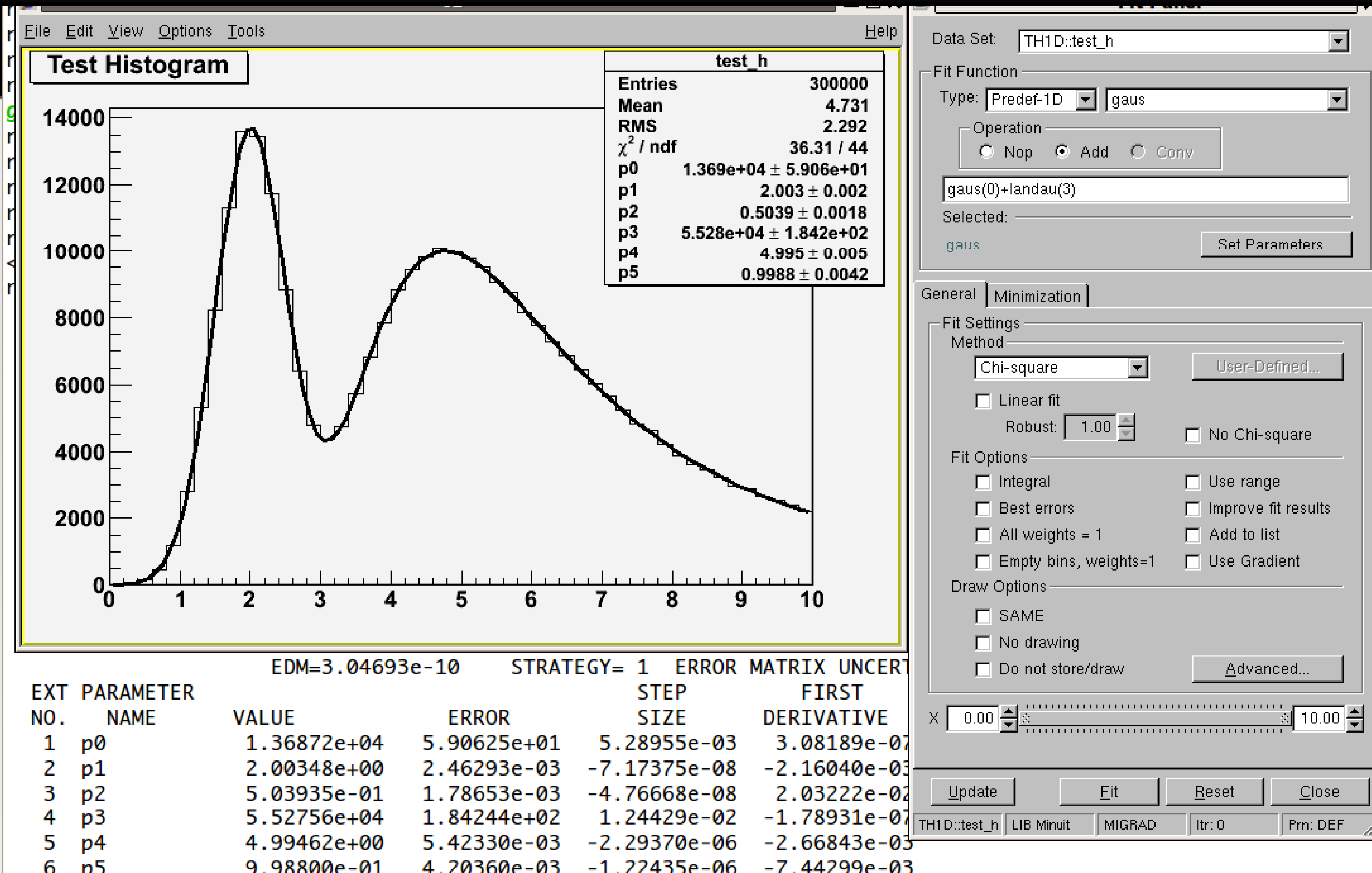
- Permit to organize 1D data in bins, or channels:
  - **TH1::Fill(val)** fills the histogram with an entry
  - **TH1::SetBinContent(bin,val)** sets the bin conten
- It also stores the expected error for each bin.
  - **TH1D *h = new TH1D("hist","Histo",50,0,10);**
  - **for (Int_t i=0; i<1e6 ; i++){**
    h->Fill(f->GetRandom(0,10));
    }
  - **h->FillRandom("test_f",1e6); // equivalent**
- Then we can display it, and fit it as well!
  - **h->Draw()**

Fitting histograms

- distribution. The method is Fit().

    ```
    TFitResultPtr fres1 =
    h->Fit("gaus","S","",0,3);
    TFitResultPtr fres2 =
    h->Fit("landau","S","",3,10);
    Double_t pars[6];
    for (Int_t i =0;i<3;i++) {
      pars[i] = fres1.Get()->GetParams()[i];
      pars[i+3] = fres2.Get()->GetParams()[i];
    }
    TF1 *f2 = new TF1("fit_f","gaus(0) + landau(3)",0,10);
    f2->SetParameters(pars);
    h->Fit(f2,"","",0,10);
    ```

Fitting an Histogram: GUI



Scatter plots

```
Int_t n = 20;
Double_t x[n], y[n]; // this works only in CINT!!!
for (Int_t i=0; i<n; i++) {
x[i] = i*0.1;
y[i] = 10*TMath::Sin(x[i]+0.2);
}
TGraph *gr1 = new TGraph (n, x, y);
```

- We can draw the graph with these options

```
gr1->Draw(”APL”)
gr1->SetMarkerStyle(20);
gr1->Draw(”APL”);
```

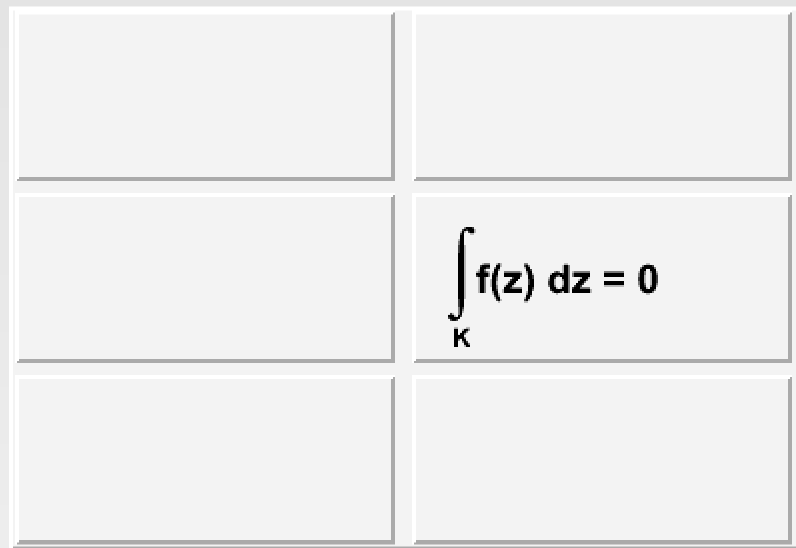
- And also fit it with a polinomial !

```
gr1->Fit("pol1","","",0.1,1.0);
gr1->Fit("pol2","","",0.1,2.0);
```

Canvases and Pads

- Single TCanvas w/ multiple TPads with TCanvas::Divide

```
root [] TCanvas c1("c1","First canvas",400,300);
root [] c1.Divide(2,3);
root [] c1.cd(4);
root [] TLatex l(0.1,0.4,"#int_{K}f(z) dz = 0");
root [] l.SetTextSize(0.25);
root [] l.Draw();
```



- The objects here are put on the ***stack*** , the part of volatile mem. that is discarded when a function returns. Does not work in macro.
- Use dynamic memory allocation: operator **new** and → to call Members

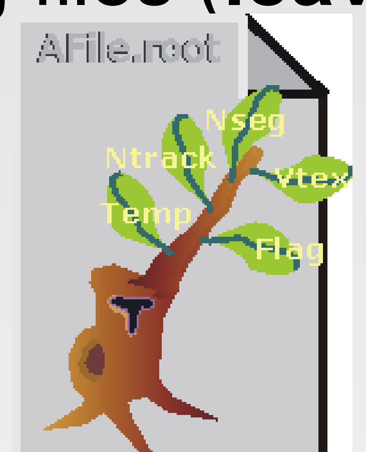
ROOT Files : TFile

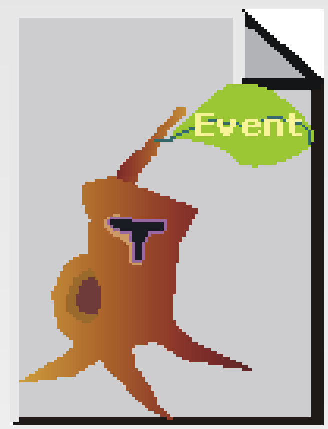
- and sub-folders.
- It can be opened Read-only (default), for writing (NEW), adding (UPDATE), rewriting (RECREATE):
  - **root [] TFile f0("file0.root",”MODE”);**
- Once opened is becomes the current directory.
- Any ROOT class object deriving from TObject can be written on the file using TObject::Write(), or Append();
- When the file is closed the contained objects are no longer available to ROOT.
- To see what is in a file TFile::GetListOfKeys()::Print()
- To retrieve an object from a File, TFile::Get(obj_name)

Data Structures in ROOT

rows="EVENTS", columns = "DATA VARIABLES"

- ROOT implements this paradigm within a more powerful interface: the **TTree**.
- Its structure is similar to that of a filesystem: it is **branched**, analogously to having directories and sub-directories, containing files (**leaves**).





TTrees in more detail

- simpler trees (**TNtuple**) have branches of sigle variables, reproducing the table paradigm.
- A branch may contain:
  - **simple variables**;
  - objects inheriting from **TObject**;
  - objects of the **TClonesArray** class (a collection of objects of the same class);
  - a **STL container** of pointers to objects.
- If it is needed a TTree can be saved on different files, and retrieved in full using one of its derivate classes: **TChain**

A very simple Tree

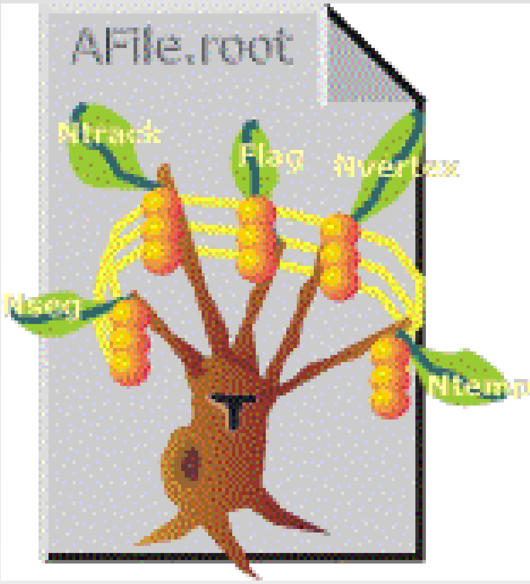
x y z
-0.676641 0.390352 0.610218
...

- The implementation is easy:
  - **root[] TTree *T = new TTree("ntuple","ascii data");**
    root[] T->ReadFile("basic.dat","x:y:z”);
- We can already draw 2-D histograms with cuts:
  - **root [] T->Draw("x:y","z>2",”lego”);**
- And save the Tree:
  - **root [] TFile *f = new TFile("basic.root","NEW");**
    root [] T->Write();

A simple Tree

- root [] Float_t x,y,z;
  root [] TTree *T2 = new TTree(”ntuple2”,”ascii2”);
  root [] T2->Branch(”x_pos”,&x,”x/F”)
  root [] T2->Branch(”y_pos”,&y,”y/F”)
  root [] T2->Branch(”z_pos”,&z,”z/F”)

- Then we fill it:
  - root [] ifstream in(”basic.dat”);
    root [] while (1) {
    in >> x >> y >> z;
    if (!in.good()) break;
    T2->Fill(); }



- And save it:
  - root [] T2->Write();
    root [] f->Close();

Reading a TTree from a TFile

- **root [] TFile f("basic.root");**
  root [] .ls

```
TFile**          basic.root
 TFile*          basic.root
  KEY: TTree  ntuple;1     ascii data
  KEY: TTree  ntuple2;1    ascii data
```

- Then create a pointer to the Tree:
  - **root [] TTree *Tr = (TTree*)f.Get("ntuple");**
    root [] Tr->GetListOfBranches().Print();

```
Collection name='TObjArray', class='TObjArray', size=16
*Br    0 :x         : x/F                                            *
*Entries :     1000 : Total  Size=      4528 bytes  File Size  =      3824 *
*Baskets :        1 : Basket Size=     32000 bytes  Compression=   1.06     *
*.........................................................................*
```

Reading a TTree from a TFile

```
root [] Float_t x,y,z;
root [] Tr->SetBranchAddress("x",&x);
root [] Tr->SetBranchAddress("y",&y);
root [] Tr->SetBranchAddress("z",&z);
```

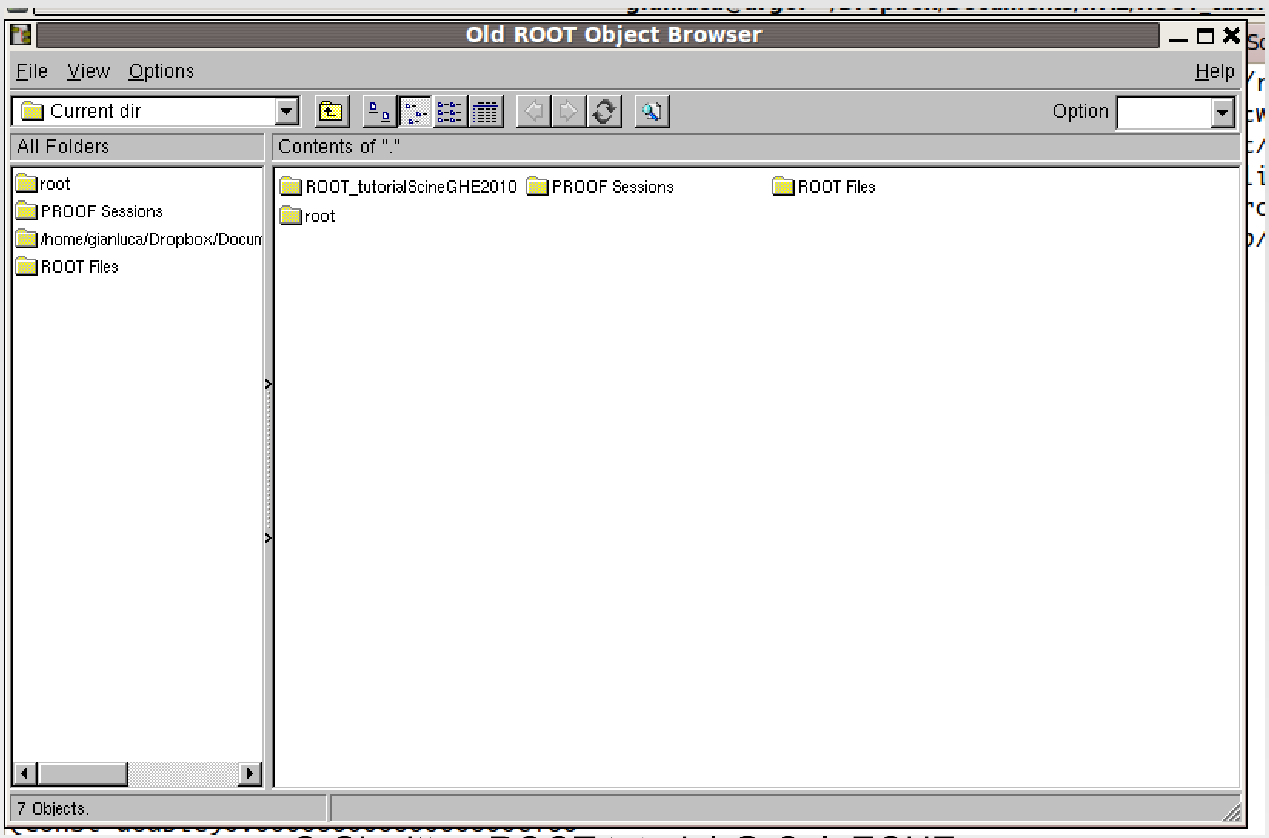
- We can then loop over the entries and get them:

```
root [] TH2D *hz =
new TH2D("x:y","x vs. y",40,-5,5,40,-5,5)
root [] Int_t ne = Tr->GetEntries();
root [] for (Int_t i=0; i<ne; i++) {
                    Tr->GetEntry();
                    if (z > 2) hz->Fill(x,y);}
root [] hz->Draw(”lego”);
```

Graphical relief

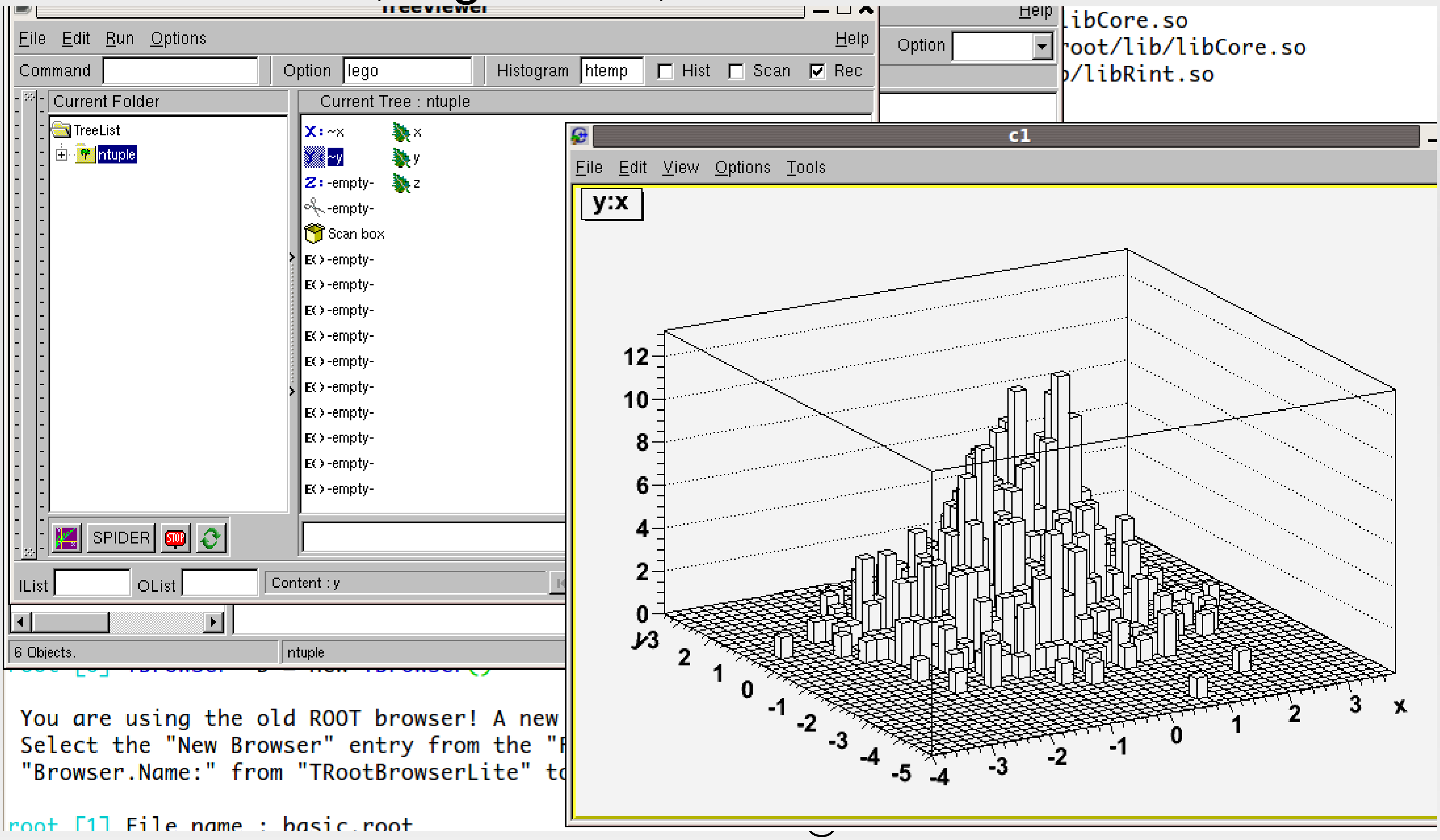
Let's have a look to what we've done:

- root [] TBrowser *b = new TBrowser();



Graphical relief II

Go to the TTree , Right Click, Start Viewer.



Using a class to fill a TTree

- down exactly to our analysis needs.
- First we need to **declare** the class:
  - **file class_tree.C:**

```
class TrackPoint : public TObject {
      public:
            Float_t x,y,z;
            TrackPoint() { x=0;y=0;z=0;}
            ClassDef(TrackPoint,1)
};
```

- Then we can write the function body, which is much like what we did before.

The macro body

```
ClassImp(TrackPoint)
void class_tree()
{
  TFile *f = new TFile("data.root","NEW");
  TTree *T = new TTree("points","ascii");
  TrackPoint *tp = new TrackPoint();
  T->Branch("tp",&tp);
  ifstream in("basic.dat");
  while (1) {
    in >> tp->x >> tp->y >> tp->z ;
    if (!in.good()) break;
    T->Fill();
  }
  T->Write();
  f->Close();
}
```

- class defin.
- TFile and TTree
- Declare class
- Branch Tree
- Read in values
- Fill Tree
- Write and Close

Running as compiled-in

- declared a class derived from a compiled one, it is necessary to run the macro as compiled code.
- This will produce a lot of errors because we did not include the proper libraries. (Interpreted CINT has that done automatically)

add to the top of class_tree.C:

```
#include "TROOT.h"
#include "TFile.h"
#include "TTree.h"
#include <iostream>
#include <fstream>

using namespace std;
```

- Then run it with:

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

Reading the Tree (again?)

It is sufficient to replace e.g. x with tp.x :

```
root [] TFile f("data.root")
Warning in <TClass::TClass>: no dictionary for class
TrackPoint is available
root [] TTree *Tr = (TTree*)f.Get("points")
root [] Tr->Draw("tp.x:tp.y","tp.z>2","lego")
```

```
root [] Tr->Draw("tp.x:tp.y","tp.z>2","box")
```

- It is also possible to copy the class definition onto another script and analyze the data.
- What if you don't have info on how the Tree was created, which classes were declared?

The TSelector framework

- It is able to recreate the classes it was created with:
  - **root [] Tr->MakeSelector()**
    Info in <TTreePlayer::MakeClass>: Files: points.h and points.C generated from TTree: points
- The two files generated must now be modified to fit our needs. A detailed walkthrough is found on http://root.cern.ch/drupal/content/accessing-ttree-tselector
  - **root [] Tr->Process(”points.C”); //or**
    root [] Tr->Process(”points.C+”);
- This harnesses the full power of ROOT.

Hands – On session: goals

- Review the examples
- Learn how to use well the documentation provided at root.cern.ch and elsewhere.
- Read into a TTree the data produced by the previous Geant 4 simulation.
  (Probably will need $ROOTSYS/tutorials/tree*.C)
- Quick-check the TTree with GUI.
- From it, construct an analysis environment with TSelector, and run it.