An Introduction to ROOT

Benno List

DESY Summer Student Tutorial
28.7.2009

ROOT

An Object-Oriented
Data Analysis Framework

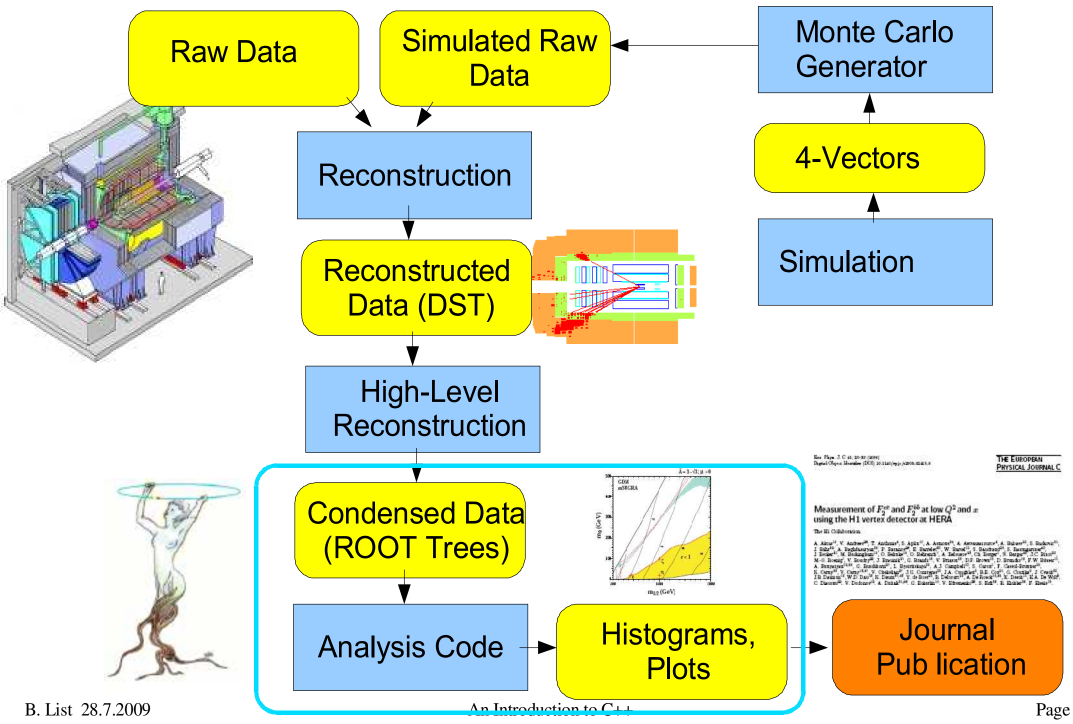
Introduction

ROOT is a Package for Data Analysis

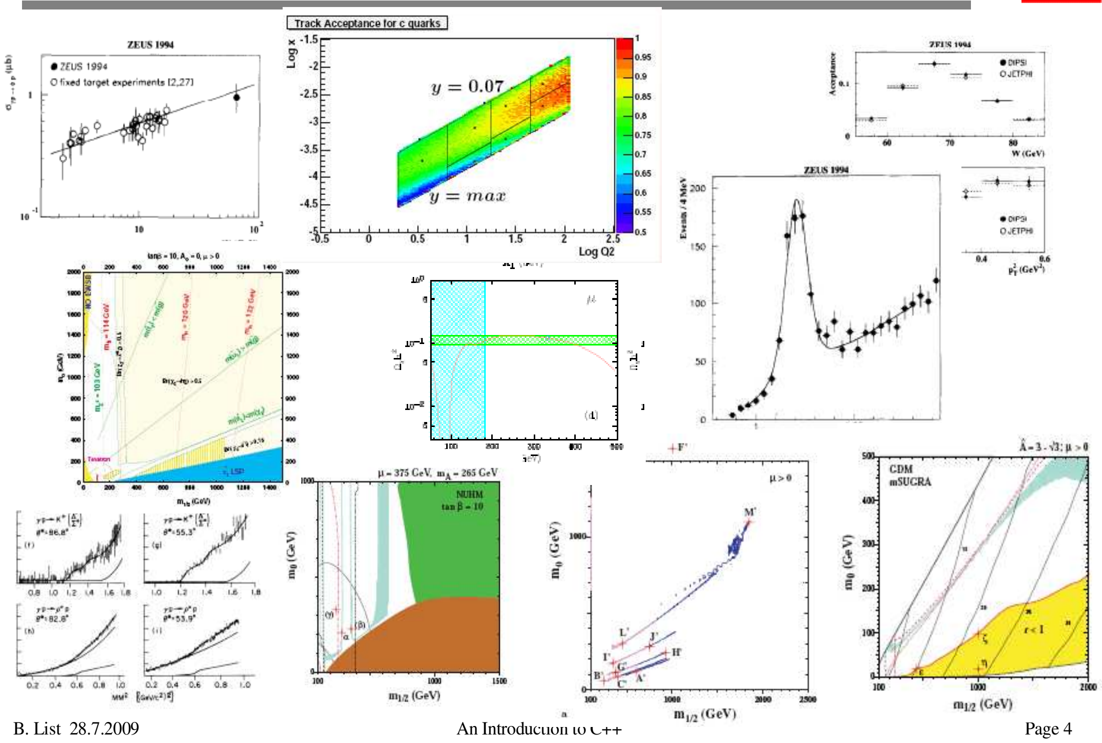
ROOT Provides:

- Several C++ Libraries
  - To store data in histograms
  - To store data in n-tuples, called “ROOT Trees”
  - To visualize histograms and n-tuples
  - To perform fits
- An Interactive Environment
  - To run C++ programs interactively
  - To visualize data
  - To perform fits

The Analysis Chain in High Energy Physics

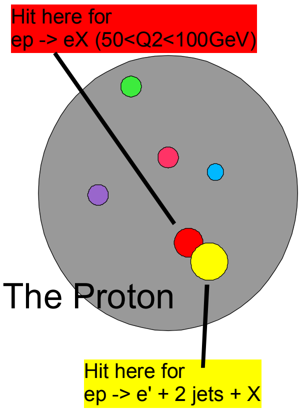


Histograms are Important in HEP



What is a Cross Section?

- Imagine small area on proton's surface
  If area σ is hit by electron, an event of a certain type happens
  Unit of σ: cm2, or barn: 1 barn = 10^{-24} cm^2 = $(10fm)^2$
  Area of proton: approx 0.02 barn (radius 0.8fm)
  Typical cross sections at HERA: pb (10^{-36} cm^2)
- Instantaneous luminosity $\mathcal{L}$:
  Number of events per second per cross section
  Unit of $\mathcal{L}$: cm^{-2} s^{-1}, or nb^{-1} s^{-1}
  HERA-II Design Lumi:
  $5{\cdot}10^{31}$ cm^{-2} s^{-1}, or 50 μb^{-1} s^{-1}
- Integrated luminosity: $\int \mathcal{L} dt$
  Number of events per cross section
  Unit of $\int \mathcal{L} dt$: cm^{-2}, or pb^{-1}
  HERA-II values: order $100pb^{-1}$



How Do we Measure a Cross Section?

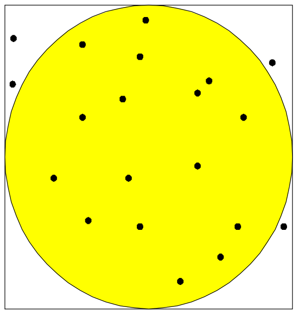
- The Master Formula:

  Number of events: $N = \sigma \cdot \int \mathcal{L} dt$

- We count events for a given data sample
  => observed number of events N_{obs}
- For this data sample, we know the integrated luminosity $\int \mathcal{L} dt$
- We are generally interested for cross sections for theoreticaly well defined processes, e.g. for ep->e' X, $0.001<x<0.002$, $5<Q^2<6 GeV^2$
- But we can only count events which we have observed, and where we have reconstructed certain x, Q^2 values, which are not exact
- => We have to correct the observed number of events for background, trigger and reconstruction inefficiencies, and resolution effects

How Do we Correct for Detector Effects?

- Analytical calculations generally not possible
- The Monte Carlo Method:
  "Generate events" randomly, which have the expected distributions of relevant properties (x, Q^2, number of tracks, vertex position...)
- Simulate detector response to each such event (hits in chambers, energy in calo)
- Pass events through same reconstruction chain as data



Measuring π with the Monte Carlo method:
The fraction f of random points within the circle is π/4.
We measure: f = 16/20 = 0.8
Uncertainty on f: sqrt(f*(1-f)/N) = 0.09
So: π/4 ~ f = 0.80 ± 0.09
andπ ~ 4f = 3.2 ± 0.3

- Now we have events where we can count events that truly fulfill our cross section criteria, and those which pass the selection criteria. The ratio is called "efficiency" and is used to correct the data

Exercise:

Write a C++ program that generates randomly points in a square and determines the fraction that lies within a circle of radius 1. From the fraction, calculate pi and its error.
How many point do you have to generate to calculate pi to a precision of 0.01, or 0.001?
How long does the program need for that?

Hint: use ROOT class TRandom (discussed later in this talk)

How Do we Count Events?

Typically: Write (and run) a program that

- Selects events with certain properties, e.g.:
  - Scattered electron with energy E'_e>10GeV
  - Tracks visible that come from a reconstructed vertex with -35<z<35cm
  - Reconstructed Bjorken-x > 0.001
- Counts events in “bins” of some quantity, e.g. Q^2:
  Q^2 = 10...20, 20...30, 30...40, ...
- Shows the number of events as a histogram

The Sketch of an Analysis Program

```
int main() {
  // some initializations here:
  //   reading steering parameters
  //   open event files

  // Book histograms

  for (int i = 0; i < events; ++i) {
    // Load event number i into memory
    // Get/calculate event properties
    if (selection_is_filfilled) {
      // fill histograms
    }
  }

  // draw the histograms
  // write out histogram file
  // write out info like number of events etc...
  return 0;
}
```

The skeleton of such an analysis program will typically be provided to you by your supervisor

Linking with ROOT

- Will normally be done by a Makefile
- Command “root-config” tells you necessary compiler flags:

```
$> root-config --incdir
/opt/products/root/5.18.00/include
$> root-config --libs
-L/opt/products/root/5.18.00/lib -lCore -lCint -lHist -lGraf
-lGraf3d -lGpad -lTree -lRint -lPostscript -lMatrix -lPhysics
-pthread -lm -ldl -rdynamic
```

- To compile a file Example.C that uses root, use:

```
$> g++ -c -I `root-config --incdir` Example.C
```

- To compile and link a file examplemain.C that uses root, use:

```
$> g++ -I `root-config --incdir` -o examplemain
   examplemain.C `root-config --libs`
```

- The inverted quotes tell the shell to run a command and paste the output into the corresponding place

ROOT Information

- Web page: http://root.cern.ch/
- We use ROOT 5.18/00
- You can download ROOT yourself and install it, also for MacOS and Windows (though I never tried it...)
- There is a User's guide at `http://root.cern.ch/drupal/content/users-guide`
- A complete overview over all classes is available at `http://root.cern.ch/drupal/content/reference-guide`

Remark: ROOT Coding Conventions

ROOT uses some unusual coding conventions
just get used to them...

- Class names start with capital T: TH1F, TVector
- Names of non-class data types end with _t: Int_t
- Class method names start with a capital letter: TH1F::Fill()
- Class data member names start with an f: TH1::fXaxis
- Global variable names start with a g: gPad
- Constant names start with a k: TH1::kNoStats
- Seperate words with in names are capitalized: TH1::GetTitleOffset()
- Two capital characters are normally avoided: TH1::GetXaxis(), *not* TH1::GetXAxis()

ROOT Histograms

- 1-Dimensional Histograms:class TH1F
  - Gives the number of entries versus one variable
  - By far the most common type
- 2-Dimensional Histograms: class TH2F
  - Gives the number of entries versus two variables
  - Used to show dependencies/correlations between variables
- Profile Histograms: class TProfile
  - Gives the average of one variable versus another variable
  - Used to quantify correlations between variables
  - Often used to quantify reconstruction resolutions/biases:
    Plot reconstructed quantity versus true (“generated”) quantity in Monte Carlo events

A 1-Dimensional Histogram Example

file gausexample.C:

```
#include <TH1.h>
#include <TFile.h>
#include <TRandom.h>

int main() {
  TH1F *histo = new TH1F ("hgaus", "A Gauss Function", 100, -5.0, 5.0);
  TRandom rnd;

  for (int i = 0; i < 10000; ++i) {
    double x = rnd.Gaus (1.5, 1.0);
    histo->Fill (x);
  }

  TFile outfile ("gaus.root", "RECREATE");
  histo->Write();
  outfile.Close();
  return 0;
}
```

Here we "book" the histogram
- ID is "hgaus" (must be unique, short, no spaces)
- Title is "A Gauss Function"
- 100 bins between -5 and 5

rnd is an object of type TRandom, a random number generator. rnd.Gaus returns a new Gaussian distributed random number each time it is called.

Open the ROOT output file
Write the histogram to it
Close the output file

Compile and run:

```
$> g++ -I `root-config --incdir` -o gausexample gausexample.C `root-config --libs`
$> ./gausexample
```


Of course, typically you will have a Makefile from your advisor which automatically links to ROOT

Exercise:

Get this program from
`/afs/desy.de/user/b/blist/public/rootintro`
Compile it and run it

What TH1F Histograms Can Do

- Booking
```
TH1F(const char* name, const char* title, int nbinsx, double xlow, double xup);
TH1F(const char* name, const char* title, int nbinsx, const double* xbins);
```
- Filling
```
virtual int Fill(double x);
virtual int Fill(double x, double w);
```
- Getting information
```
virtual double GetBinContent(int bin) const;
virtual double GetMaximum(double maxval = FLT_MAX) const;
virtual double GetMaximum(double maxval = FLT_MAX) const;
```
- Adding etc.
```
virtual void Add(TF1* h1, Double_t c1 = 1, Option_t* option);
```
likewise: `Multiply, Divide`
- Drawing
```
virtual void Draw(Option_t* option);
```
- Writing to a file (inherited from TObject)
```
virtual int Write(const char* name = "0", int option = 0, int bufsize = 0);
```


For detailed information, look at

http://root.cern.ch/root/html518/TH1F.html
http://root.cern.ch/root/html518/TH1.html
http://root.cern.ch/root/html518/TObject.html

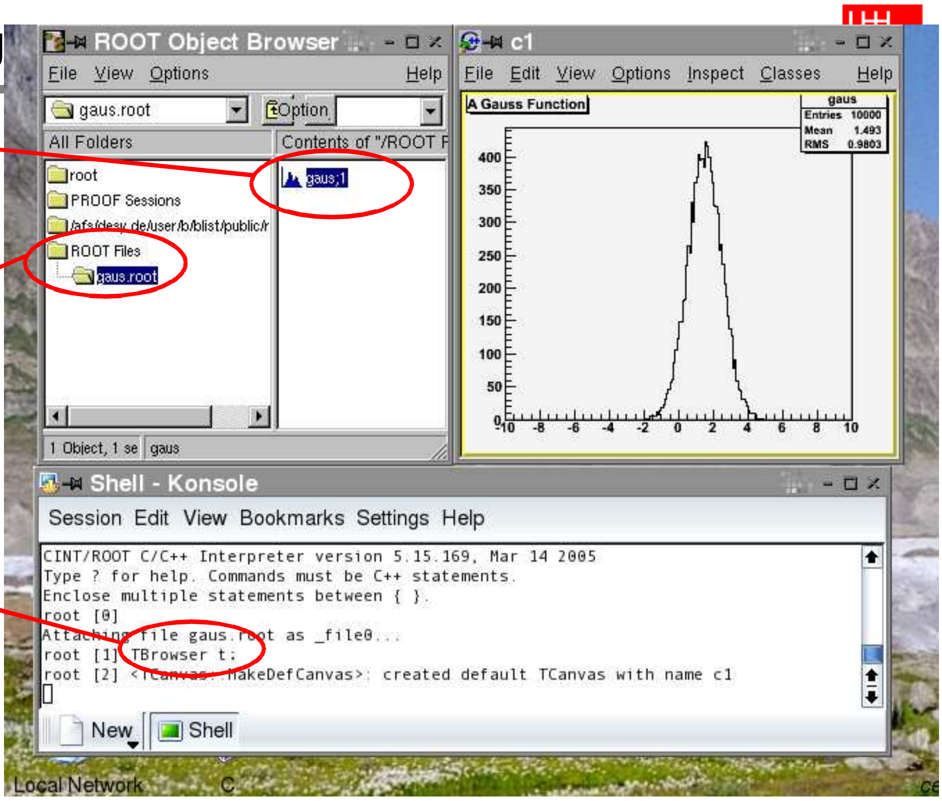
Looking at the Histogram: Interactive ROOT

- Start ROOT interactively with
  `$> root`
- A DESY specialty: You can chose a special ROOT version with
  `$> ini ROOT40008`
  (other versions: ROOT40402, ROOT51200 etc)
- At the ROOT prompt, enter
  `root [1] TBrowser t;`
- this opens a browser

Clicking

Click here to display a histogram

Click here to open a file



Enter this to get the browser window

No Clicking

```
$> root

root [0] TFile *file0 = TFile::Open("gaus.root")
root [1] hgaus.Draw()
root [2] hgaus.Draw(“E”)
root [3] hgaus.Draw(“C”)
root [4] gStyle->SetOptStat(1111111)
root [5] hgaus.GetXaxis()->SetTitle("Abscissa")
root [6] hgaus.GetYaxis()->SetTitle("Ordinate")
root [7] gPad->SetLogx(1)
root [8] hgaus.Draw(“E2”)
root [9] hgaus.SetLineColor(3)
root [10] hgaus.SetLineStyle(2)
root [11] hgaus.SetLineWidth(2)
root [12] hgaus.SetMarkerStyle(20)
root [13] hgaus.SetMarkerSize(1.5)
root [14] hgaus.SetMarkerColor(4)
root [15] hgaus.Draw(“E1”)
root [16] hgaus.SetFillColor(4)
root [17] hgaus.Draw(“C”)
root [18] gPad->Print(“gaus1.ps”)
root [19] .q
```


From ROOT manual, Section 3 (Histograms):

Statistics Display

By default, drawing a histogram includes drawing the statistics box. To eliminate the statistics box use: TH1::SetStats(kFALSE).
If the statistics box is drawn, you can select the type of information displayed with gStyle->SetOptStat(mode). The mode has up to seven digits that can be set to on (1) or off (0). mode = iourmen (default = 0001111)

- n = 1 the name of histogram is printed
- e = 1 the number of entries printed
- m = 1 the mean value printed
- r = 1 the root mean square printed
- u = 1 the number of underflows printed
- o = 1 the number of overflows printed
- i = 1 the integral of bins printed

WARNING: never call SetOptStat(000111); but SetOptStat(1111), 0001111 will be taken as an octal number.

Drawing Options for 1D-Histograms

"AXIS"	Draw only axis
"AH"	Draw histogram, but not the axis labels and tick marks
"]["	When this option is selected the first and last vertical lines of the histogram are not drawn.
"B"	Bar chart option
"C"	Draw a smooth Curve througth the histogram bins
"E"	Draw error bars
"E0"	Draw error bars including bins with o contents
"E1"	Draw error bars with perpendicular lines at the edges
"E2"	Draw error bars with rectangles
"E3"	Draw a fill area througth the end points of the vertical error bars
"E4"	Draw a smoothed filled area through the end points of the error bars
"L"	Draw a line througth the bin contents
"P"	Draw current marker at each bin except empty bins
"P0"	Draw current marker at each bin including empty bins
"*H"	Draw histogram with a * at each bin
"LF2"	Draw histogram like with option "L" but with a fill area. Note that "L" draws also a fill area if the hist fillcolor is set but the fill area corresponds to the histogram contour.

For detailed information, look at
`http://root.cern.ch/root/html518/THistPainter.html`

Color and Color Palettes

At initialization time, a table of basic colors is generated when the first Canvas constructor is called. This table is a linked list, which can be accessed from the `gROOT` object (see `TROOT::GetListOfColors()`). Each color has an index and when a basic color is defined, two "companion" colors are defined:

- The dark version (`color_index + 100`)
- The bright version (`color_index + 150`)

The dark and bright colors are used to give 3-D effects when drawing various boxes (see `TWbox, TPave, TPaveText, TPaveLabel`, etc). If you have a black and white copy of the manual, here are the basic colors and their indices:

1 = black
2 = red
3 = bright green
4 = bright blue
5 = yellow
6 = hot pink
7 = aqua
8 = green
9 = blue

0 -> 9: basic colors
10 -> 19: gray shades
20 -> 29: brown shades
30 -> 39: blue shades
40 -> 49: red shade

The list of currently supported basic colors (here dark and bright colors are not shown) are shown. The color numbers specified in the basic palette, and the picture above, can be viewed by selecting the item "`Colors`" in the "`View`" menu of the canvas toolbar. Other colors may be defined by the user. To do this, one has to build a new `TColor` object:

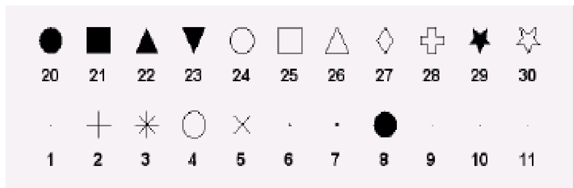
ROOT manual, Section 9: Graphics and Graphical User Interfaces

Drawing Options for 2D-Histograms

AXIS	Draw only axis
ARR	arrow mode. Shows gradient between adjacent cells
BOX	a box is drawn for each cell with surface proportional to contents
COL	a box is drawn for each cell with a color scale varying with contents
COLZ	same as "COL". In addition the color palette is also drawn
CONT	Draw a contour plot (same as CONT0)
CONT0	Draw a contour plot using surface colors to distinguish contours
CONT1	Draw a contour plot using line styles to distinguish contours
CONT2	Draw a contour plot using the same line style for all contours
CONT3	Draw a contour plot using fill area colors
CONT4	Draw a contour plot using surface colors (SURF option at theta = 0)
CONT5	Draw a contour plot using Delaunay triangles
LIST	Generate a list of TGraph objects for each contour
FB	Draw current marker at each bin including empty bins
BB	Draw histogram with a * at each bin
SCAT	Draw a scatter-plot (default)
TEXT	Draw bin contents as text
TEXTnn	Draw bin contents as text at angle nn (0 < nn < 90)
[cutg]	Draw only the sub-range selected by the TCutG named "cutg"

One Point, or Marker: Class **TMarker**

A marker is a point with a fancy shape! The possible markers are the following:



ROOT Manual, Section 9: Graphics and Graphical User Interfaces

CINT

- ROOT uses a C++ interpreter CINT for interactive use
- You can enter any C++ command; trailing ";" is not required
- Resetting the interpreter (erasing variables etc):
  `root[] gROOT->Reset()`
  Do that often! But often a restart of ROOT is needed...
- Special commands:

`.q`	Quit
`.x script.C`	Execute script "script.C"
`.L script.C`	Load script "script.C" (if script.C contains class definitions)

- More in Chapter 7: "CINT the C++ Interpreter" of ROOT manual

Two kinds of scripts

- Un-named scripts:

```
{
  #include <iostream.h>
  cout << "Hello, World!\n";
}
```

  – Code must be enclosed in curly braces!

  – Execute with

  ```
  root[] .x script.C
  ```

- Named scripts:

```
#include <iostream.h>
int main() {
  cout << "Hello, World!\n";
}
```

  – More like normal C++ programs, recommended form!

  – Execute with:

  ```
  root[] .L script.C
  root[] main()
  ```

CINT Extensions to C++

- If you create a pointer and assign to it with “new”, you don't need to declare the pointer type:
```
h = new TH1F (“h”, “histogram”, 100, 0, 1)
```
  - h is automatically of type TH1F*
- “.” can be used instead of “->”
  => Don't do that habitually!
- If you use a variable that has not been declared earlier, ROOT tries to create one for you from all named objects it knows
  => If you have opened a file that contains a histogram “hgaus”, you can directly use
```
hgaus->Draw()
```
  - But be careful: Sometimes you get a different object than you thought :-(

TF1 Functions and Fitting

file tf1example.C:

```
#include <TH1F.h>
#include <TF1.h>
#include <TFile.h>

Double_t mygauss (Double_t *x, Double_t *par) {
  // A gauss function, par[0] is integral, par[1] mean, par[2] sigma
  return 0.39894228*par[0]/par[2]*exp(-0.5*pow(( *x -par[1])/par[2], 2));
}

int main() {
  TF1 *gaussfun = new TF1 ("gaussfun", mygauss, -10, 10, 3);
  gaussfun->SetParameters (100, 0., 1.);
  gaussfun->SetParNames ("Area", "Mean", "Sigma");
  TFile *file = new TFile ("gaus.root");
  TH1F *hgaus = dynamic_cast<TH1F *>(file->Get("hgaus"));
  if (hgaus) {
    hgaus->Fit(gaussfun);
  }
}
```

Defines a Gauss function
Note that the argument must be handed over by a pointer!!!

Defines a TF1 function object
- ID is “gaussfun”
- It executes function mygauss
- It is valid for x between -10 and 10
- It has 3 parameters

Here we load the histogram “hgaus” from the file “gaus.root”, and if it was found, we fit it.

`file->Get()` returns only a pointer to a `TObject`, which is a base class of `TH1F`.
With `dynamic_cast` we convert the pointer to the correct type.
If the object pointed to is not a `TH1F` (it could something completely different!), the `dynamic_cast` returns a null pointer.

Exercise:

Get this program from
`/afs/desy.de/user/b/blist/public/rootintro`
Compile it and run it with

```
g++ -I `root-config --incdir` -o tf1example tf1example.C `root-config
--libs`
./tf1examaple
```

Run it in root interactively with

```
$> root
root [0] .L tf1example.C
root [1] main()
```

Learn more about TF1 and fitting in Chapter 5 “Fitting Histograms” of the ROOT manual.

Five Minutes on ROOT Trees

- A ROOT Tree holds many data records of the same type, similar to an n-tuple. One record is described by a C++ Class:

```
class EventData {
  public:
    Int_t run;
    Int_t event;
    Float_t x;
    Float_t Q2;
};
```

- The ROOT Tree knows how many enries (here: events) it contains.
  It can fill one instance (one object) of class EventData at a time with data, which we then can use to plot the data.

```
TH1F *histox = new TH1F ("histox", "Bjorken x", 1000, 0., 1.);
TFile *file ("eventdata.root");
TTree *tree = dynamic_cast<TTree *>(file->Get("eventdata"));
EventData *thedata = new EventData;
TBranch *branchx = tree->GetBranch("x");
branchx->SetAddress (&(event->x));
for (int i = 0; i < tree->GetEntries(); ++i) {
  branchx->GetEntry(i);
  histox->Fill (x);
}
```

Trees, Branches, and Leaves

- The Tree is the whole data set
- A Branch contains the data of one or several variables, e.g. the x and Q2 values of all events.
  - A Tree consists of several Branches.
  - How the Branches are set up is determined by the program that writes the Tree
- A Leaf is the data of a single variable (like x)
  - A Branch consists of several Leaves

Using Trees

- You will surely given a program by your advisor which reads in a ROOT Tree so don't worry how to create a ROOT Tree.
- You will have an “event loop” which loops over all entries of the tree. Within the loop, you'll find all data that you need in some object.
- Use this data to select “good” events and plot their properties in histograms