

Introduction to ROOT

ROOT: object-based development environment dedicated to data analysis, and based on C++

www : root.cern

Modes of work

- Interactive (terminal session)
 - online-interpretted C++ commands
 - Macros (interpreted or compiled), two possible modes:
 - simplified: string of commands
 - within functions ; more conformity to C++ standard required
- As compilable C++ code : using Root libraries

Includes:

- Graphical windows, histograms, scatter plots (with uncertainties)
- Mathematical function (including special functions) : drawing, evaluation, pulling from distributions
- Fitting of functions to data,
- Data bases (“trees”). Data filtering (“cuts”)
- Pulling from random distributions
- Collections of objects, I/O with storage of objects
- Numerical algorithms. Spectrum analysis. DataFrame-type bases. Machine learning. GUI.



Guides, manuals, help

- Help TOC: root.cern/get_started/
- Manual: root.cern/manual/
- Primer: root.cern/primer/
- Slides indico.cern.ch/event/395198/attachments/791523/1084984/ROOT_Summer_Student_Tutorial_2015.pdf
- Forum: root-forum.cern.ch
- Documentation: root.cern/doc/master/

Notice: class names in this script are clickable and point to relevant ROOT help pages :)

Installation:

- Download: root.cern/install/#download-a-pre-compiled-binary-distribution
sources or binaries for: Linux, Windows, Mac
- Linux: Go to your login script (`~/.bash_login` or `~/.bashrc`) and add line:

```
. [your_ROOT_path]/bin/thisroot.sh
```
- Quantum jump:

Versions ≤ 5.34	vs	Versions ≥ 6.00
“cint” interpreter		“cling” interpreter
Syntax tolerance		Syntax rigor
Standard: ~ c++98		Standard ~ c++11

MODE I: Interactive session ROOT as a calculator and interpreter of C++ commands

- Launching:
`root`
`root -l` (without 'welcome' splash screen)
`root -b` (without graphics, but faster instead)
- Inside session. Quitting: `.q`
Shell command: `!. [command]`
Executing a macro: `.x`
Forced exit: `.qqqqqqqq`

```
root [0] sqrt (1.23)
(const double)1.10905365064094164e+00
root [1] double x = pow (sin(0.5),2.) + pow (cos(0.5),2.)
root [2] x
(double)1.000000000000000000e+00
root [3] cout << x << endl
1
(class ostream)139768533438272
```

Data types

① C++ types: `int, double, char text[100], string napis,`
`vector<double> vec, double* d, int& i, ...`

② Internal (ROOT) types, as overlays on the C++ types:

`Int_t , Float_t , Double_t, Char_t, Bool_t`

Motivation: making the code machine-indepent

List of machine-independent types:

root.cern/root/html/doc/guides/users-guide/ROOTUsersGuide.html#machine-independent-types

TMath

Mathematical class

(Notice: class names are links to help sites)

- Functions (TMath::Sqrt(x) , Power, SinH, Exp, Gaus, Factorial, ...)
- Mathematical and physical constants (TMath::Pi(), E, RadToDeg, DegToRad, Hbar, K)
- Operations (TMath::Abs(x) , Min, Max, ...)
- Special functions (TMath::BesselI (x) , BesselJ/K/Y , Erf, ...)

```
root [0] TMath::Power (TMath::Pi() , 1./3.)  
(Double_t)1.46459188756152314e+00
```

Notice: The [ROOT::Math](#) namespace contains even more functions and algorithms



Autocompletion – intelligent handy help

```
root [1] TMath::Pi
```



```
Pi
```

```
PiOver2
```

```
PiOver4
```

```
root [1] TMath::Pi(
```



```
Double_t Pi()
```

```
root [1] TMath::Power(
```



```
LongDouble_t Power(LongDouble_t x, LongDouble_t y)
```

```
LongDouble_t Power(LongDouble_t x, Long64_t y)
```

```
LongDouble_t Power(Long64_t x, Long64_t y)
```

```
Double_t Power(Double_t x, Double_t y)
```

```
Double_t Power(Double_t x, Int_t y)
```

TCanvas

Graphical window

```
root[0] new TCanvas  
(class TCanvas*)0x2c1e5e0  
root[1] c1->Set
```

← “quick” creation of graphical window with generic properties

← Notice: automatic name assignment (“c1”)



```
root[1] c1->SetTitle ("HelloCanvas")  
root[2] c1->GetTitle ()  
(const char* 0x1557339) "HelloCanvas"  
root[3] c1->ls ()  
root[4] c1->Close ()
```

root[5] TCanvas c2 ← We create a new window. Before we used pointer. Now – object.

```
root[6] c2.GetName()  
(const char* 0x16632b1) "c1_n2" ← Title is different than variable name!
```

```
root[7] TCanvas c3 (
```



Multitude of constructors

```
TCanvas TCanvas(Bool_t build = kTRUE)  
TCanvas TCanvas(const char* name, const char* title = "", Int_t form = 1)  
TCanvas TCanvas(const char* name, const char* title, Int_t ww, Int_t wh)  
TCanvas TCanvas(const char* name, const char* title, Int_t wtopx, Int_t wtopy,  
Int_t ww, Int_t wh)  
TCanvas TCanvas(const char* name, Int_t ww, Int_t wh, Int_t winid)  
root[7] TCanvas c3 ("c3canvas", "My canvas", 600, 400);
```

↑
Proper name
of object
(within C++).

↑
"Name"
(identifier)
(within ROOT).

↑
Displayed title
(just a c-string)

```
root[8] c3Canvas ← Name as identifier or replacement of C++ name  
(class TCanvas*)0x16964d0
```

```
root[9] TCanvas* c4 = new TCanvas ("c4canv", "2nd Canvas", 600, 400);
```



↑ Dynamic allocation (we then use a pointer to an object)

TFn $n = \{1, 2, 3\}$

Functions

```
root[0] TF1 f1 ("f1", "sin(x)/x", 0. , 10. );
root[1] f1.Draw ()
root[2] f1.SetRange (-10. , 10.)
root[3] f1.Draw ()
root[4] f1.Eval (1.) or f1(1.)
root[5] f1.Integral ( 0. , TMath::Pi() )
root[6] f1.GetMinimum ( 1e-10 , 5.)
```

← function with given formula and range

```
root[7] TF1 fb ("fb", "[0]*sin([1]*x)/x", 0., 10.);
root[8] fb.SetParameter (0, 0.5);
root[9] fb.SetParameter (1, 2. );
root[10] fb.Draw ("same")
```

← parameter-dependent function

```
root[11] fb.SetLineColor (2);
```

40	41	42	43	44	45	46	47	48	49
30	31	32	33	34	35	36	37	38	39
20	21	22	23	24	25	26	27	28	29
10	11	12	13	14	15	16	17	18	19
0	1	2	3	4	5	6	7	8	9

Color numbering scheme for the generic palette

```
root[12] f1.SetLineWidth (2);
root[13] f1.SetLineStyle (2);
root[14] f1.Draw ("same");
```

10	— · — · — · — · — · — · — · — · — ·
9	— — — — — — — — — — — — — — — —
8	····· ····· ····· ····· ····· ····· ·····
7	- - - - - - - - - - - - - - - - - - -
6	- · - · - · - · - · - · - · - · - ·
5	- · - · - · - · - · - · - · - · - ·
4	- · - · - · - · - · - · - · - · - ·
3	····· ····· ····· ····· ····· ····· ·····
2	- - - - - - - - - - - - - - - - - - -
1	— — — — — — — — — — — — — — — —

Numbering scheme of line styles

```
root[15] TF2 f3 ("f3", "exp( -(x-0.5)*(x-0.5)/0.05 - (y-0.5)*(y-0.5)/0.05) ");  
root[16] f3.Draw ()  
root[17] f3.Draw ("lego2")  
root[18] f3.Draw ("colz")
```

← Graphical options, e.g.: "surf", "surf2", "cont"

root.cern/doc/master/classTHistPainter.html#HP01

(Notice: more methods of declaring functions are available in C++ macros.)

Writing down a graphical window in a file:

```
root[1] c1->Print ("picture.ext");
```

ext = {gif, jpg, pdf, png, ps, svg, root, tex, tiff, xml, xpm, C}

Caution: lossy -vs- lossless formats of graphics storage

```
root[2] c1->Clear ();
```

TGraph

Scatter plots of data points

(and related ones: [TGraphErrors](#) / [TGraphAsymmErrors](#) / [TGraphBentErrors](#))

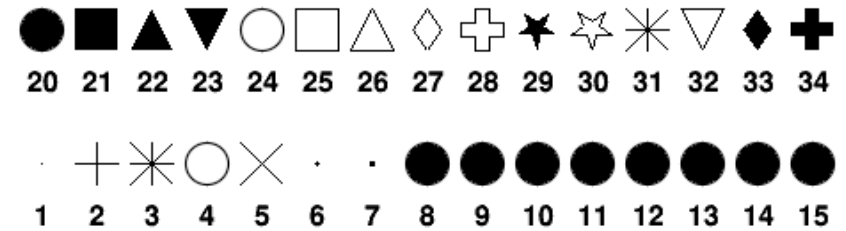
tgraph.dat x

```
0.0 0.01
0.1 0.98
0.2 2.03
0.3 2.99
0.4 4.07
0.5 5.01
```

In a Linux console: `wget www.fuw.edu.pl/~kpias/ctnp/tgraph.dat`

```
root[1] TGraph g ("tgraph.dat")
root[2] g.Draw ("AP")

root[3] g.SetMarkerSize (0.8)
root[4] g.SetMarkerStyle (20)
root[5] g.Draw ("AP")
```



Marker styles

You can select which columns to read:

`wget www.fuw.edu.pl/~kpias/ctnp/tgraph2.dat`

```
root[1] TGraph g ("tgraph2.dat", "%*s %lg %*lg %lg");
root[2] g.Draw ("AP");
```

```
root[3] TGraph gr2 (
(...)
```

```
TGraph TGraph (Int_t n, const Double_t* x, const Double_t* y)
```

```
TGraph TGraph (Int_t n, const Float_t* x, const Float_t* y)
```

```
TGraph TGraph (Int_t n, const Int_t* x, const Int_t* y)
```

```
(...)
```

```
root[4] Double_t x[] = {0.05, 0.95, 1.95, 2.05, 3.05, 3.95};
```

```
root[5] Double_t y[] = {1.00, 1.11, 1.29, 1.41, 1.52, 1.59};
```

```
root[6] TGraph gr2 (6, x, y);
```

```
root[7] gr2.SetMarkerSize (0.8) ; gr2.SetMarkerStyle (24)
```

```
root[8] gr2.SetMinimum (0.) ; gr2.SetMaximum (5.5);
```

```
root[9] gr2.GetXaxis()->SetLimits (0, 5);
```

```
root[10] gr2.Draw ("AP")
```

```
root[11] gr.Draw ("sameP")
```

%lg : read column as double precision
%*lg: column type is double; omit it.
%*s : column type is string; omit it.

TRandomN , $N = \{ 1, 2, 3 \}$ Pseudorandom numbers

► Usage of [TRandom3](#) is advised in the documentation. Calling time ~ 45 ns. Period $\sim 10^{6000}$.

```
root[1] TRandom3 r;  
root[2] r.SetSeed ();
```

← setting the pseudorandom seed

► TRandomN have predefined distributions, e.g. :

```
Binomial (ntot, prob)  
Exp (tau)  
Landau (mean, sigma)  
Rndm ()       ← returns double  $\in [0, 1)$ 
```

```
BreitWigner (mean, gamma)  
Integer (imax)  
Gaus (mean, sigma)  
Poisson (mean)
```

```
root[3] r.Rndm ()  
(Double_t) 9.997417e-01  
root[4] r.Gaus (15.3, 0.02)  
(Double_t) 1.531998e+01
```

► Moreover, you can pull numbers from distribution defined by the user in form of TF1 object, e.g. :

```
root[5] TF1 fun1 ( "fun1", "x*x*exp(-x/0.5)" , 0., 5.) ;  
root[6] fun1.GetRandom ()  
(Double_t) 6.916067e-01
```

► You can also pull a number from an user-defined histogram (description of histograms – soon).

TVectorN , $N = \{2, 3\}$ 2-3 dimensional vectors

```
root[1] TVector3 v1 (1, 2, 3) , v2, v3;
root[2] v2.SetXYZ (-1,-2,-3);
```

v1.Mag() Mag2() Theta() CosTheta() Phi() Perp() ← basic properties

```
root[3] v3 = -3. * (v1 + v2);    v3 -= 2. * v1;    v3.Print()
```

```
root[4] v1.Cross(v2) .Print();    ← cross product
```

```
root[5] v1.Dot ( v3.Orthogonal () )    ← dot product; perpendicular vector
```

v1.Angle (v2); v1.RotateX/Y/Z (angle); v3.Rotate (angle, v2)

TLorentzVector Four-vector

Has 4 dimensions, that you can use either as $[X, Y, Z, T]$, or $[P_x, P_y, P_z, E]$. (Caution: sequence!)

It is implemented as $\text{TVector3} \oplus \text{double}$.

```
root[1] TLorentzVector L (1, 2, 3, 4); cout << L.T() << endl;
```

L.Pt() P()

← $\sqrt{P_x^2 + P_y^2}$, $|\vec{P}|$

L.M()

← $\pm \sqrt{\text{spacetime interval}} / \pm \text{available energy} / \pm \text{invariant mass}$

L.Beta() Gamma()

← $\beta = \pm |\vec{P}|/E$, $\gamma = 1/\sqrt{1 - \beta^2}$

```
root[2] TLorentzVector v4piplus (0., 0., 1., sqrt(1*1 + 0.1395*0.1395) );
```

```
root[3] v4piplus.Rapidity();    ←  $y = 0.5 \cdot \ln [(E-p_z)/(E+p_z)]$ 
```

```
root[4] TVector3 beta (0., 0., 0.7);
```

```
root[5] v4piplus.Boost (beta);    ← Lorentz transformation by “beta” velocity vector
```

THdt, $d = \{1, 2, 3\}$
 $t = \{C, S, I, F, D\}$

Histograms

Type of variables that stores the bin content

Dimension of histogram

Type	C	S	I	F	D
Max bin content	$2^8 - 1$	$2^{16} - 1$	$2^{31} - 1$		
Max precision				7 digits	14 digits

```

root[1] TH1F h1 ("hist1", "My histogram", 100, -10., 10.);
root[2] h1.Fill (5.23);
root[3] h1.Draw ();
root[4] h1.Fill (3.21, 0.1); h1.Draw ();
root[5] h1.Draw ("hist");

root[6] TRandom3 r; r.SetSeed ();
root[7] for (int i=0; i < 1e5 ; i++) h1.Fill ( r.Gaus() );
root[8] h1.Draw ();

```

No of bins From To
 ▼ ▼ ▼
 Fill the bin containing $x = 5.23$, with weight of 1.
 Fill the bin containing $x = 3.21$, with weight of 0.1.

Drawing options:

For TH1_: "same", "e", "e0 ... 4", "hist", ...

For TH2_: "box", "col", "cont", "lego", "surf" (+ other options, e.g. "lego2")

→ root.cern/root/html/doc/guides/users-guide/ROOTUsersGuide.html#draw-options

Options can be combined, e.g. `h.Draw ("colz") , h.Draw ("elsame")`

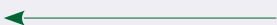
Histograms, cont.

```
root[1] h1.Set 
```

```
root[1] h1.SetTitle ("New title")
```

```
root[2] h1.SetMinimum (-20) ; h1.SetMaximum (200)
```

```
root[3] h1.SetLineColor (...) / SetLineStyle (...) / SetLineWidth (...)
```

```
root[4] h1.SetStats ( 0 / 1 );  (Don't) plot stats (for details, see description of TStyle)
```

```
h1.SetNdivisions ( code , " Axis " );
```

Code = $N1 + 100 \cdot N2 + 10000 \cdot N3$, where:

N1: (expected) number of leading divisions

N2: (expected) number of 2. rank divisions

N3: (expected) number of 3. rank divisions

```
root[4] h1.Get 
```

```
root[4] cout << h1.GetMean() << '\t' << h1.GetRMS() << '\t' << h1.GetNbinsX();
```

```
root[5] h1.Integral ( 
```

```
Double_t Integral(Int_t binx1, Int_t binx2, Option_t* option = "") const
```

We encounter a problem. How to find the bin no. that corresponds to given x ? → `FindBin` (x) method

```
root[6] h1.Integral (h1.FindBin (-2.) , h1.FindBin(2.) )
```

Caution: Integral without specifying option – does not integrate ($\sum h_i \cdot \Delta$) but sums up the contents of bins ($\sum h_i$). In order to calculate integral, you have to type "width" option.

Histograms, cont.

► Access to histogram data.

Bin numbering convention: [1 ... h1.GetNbinsX()]

```
root[1] cout << h1.GetBinContent (41) << '\t' << h1.GetBinError (41) << endl;
11 3.31662
```

As you can see, uncertainties are Poissonian

```
root[2] Float_t* hy = h1.GetArray ()
```

Will return the array with bin contents

Attention for the numbering: hy[1] ... hy[N]

In hy[0] , hy[N+1] the *underflow / overflow* are stored

► Unfortunately, there is no method directly extracting the array of uncertainties. Instead you can:

```
root[3] Float_t* hyerr = new Float_t [h1.GetNbinsX() + 2]
root[4] for (int i=0; i<= h1.GetNbinsX()+1; i++) hyerr[i] = h1.GetBinError(i)
```

► You can also extract the array of positions of centers of bins:

```
root[5] Float_t* hx = new Float_t [h1.GetNbinsX() + 2]
root[6] for (int i=0; i<= h1.GetNbinsX()+1; i++) hx[i] = h1.GetBinCenter (i)
```

► To create a separate data structure for uncertainties, issue this before filling the histogram:

```
root[7] h1.Sumw2 ()
```

Histograms, cont.

Operations on histograms (Addition, multiplication, division)

```
root[1] TH1F::Add ( 
```

```
Bool_t Add(TF1* h1, Double_t c1 = 1, Option_t* option = "")
```

```
Bool_t Add(const TH1* h, const TH1* h2, Double_t c1 = 1, Double_t c2 = 1)
```

```
Bool_t Add(const TH1* h1, Double_t c1 = 1)
```

*this = c1*h + c2*h2*

*this += c1*h1*

```
root[2] TH1F h2 (h1); h2.Reset() ← New histogram: such as h1, but empty.
```

```
root[3] for (int i=0; i < 1e5 ; i++) h2.Fill ( r.Gaus (5., 1.) ); ← Filling with Gauss distrib.
```

```
root[4] h2.Draw(); h2.Add ( &h1 , 0.1 ); h2.Draw ("e1") ← We are adding
```

```
root[5] TH1F h3 (h1); h3.Reset()
```

```
root[6] for (int i=1; i<1000; i++) h3.Fill (-9.99 + 0.02*i) ; ← Uniform distribution
```

```
root[7] h2.Multiply ( &h3 )
```

We are multiplying

```
root[8] h2.Draw ("e1")
```

Uncertainties follow the propagation formula

```
root[9] TH1F h4 (h1); h4.Reset ()
```

```
root[10] h4.Divide ( &h2 , &h3 )
```

We are dividing

```
root[11] h4.Draw ("e1")
```

Uncertainty arise, not cancel out, as expected

2-Dim Histograms

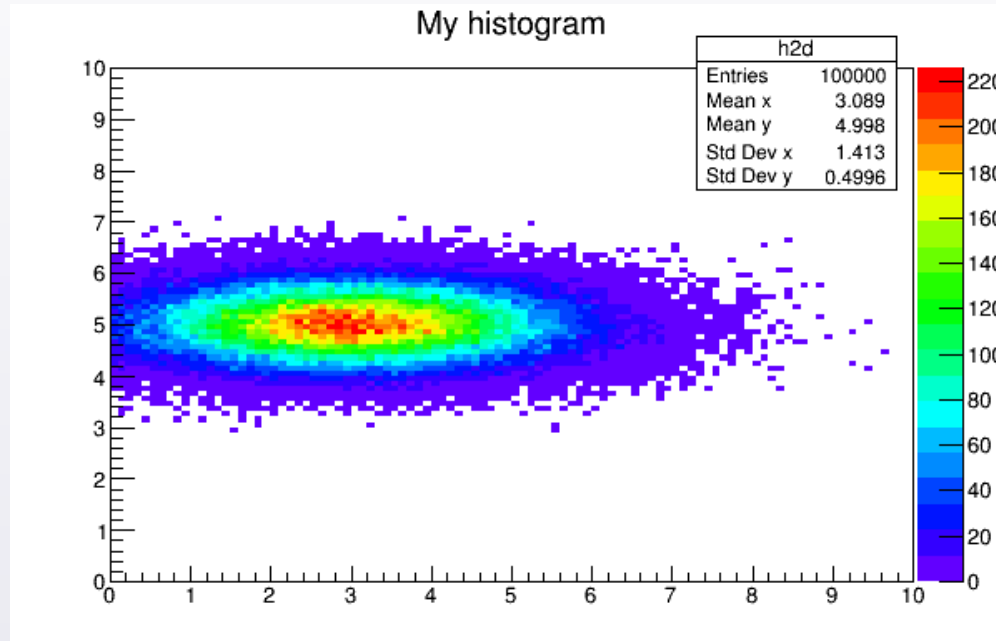
X AXIS

No of bins From To
▼ ▼ ▼

Y AXIS

No of bins From To
▼ ▼ ▼

```
root[1] TH2F h2d ("h2d", "My histogram", 100, 0., 10., 100, 0., 10.);
root[2] for (int i=0; i<1e5; i++) h2d.Fill (r.Gaus(3,1.5), r.Gaus(5,0.5));
root[3] h2d.Draw ("colz")
```



```
root[4] cout << h2d.GetNbinsX() << '\t' << h2d.GetNbinsY() << endl;
100      100
```

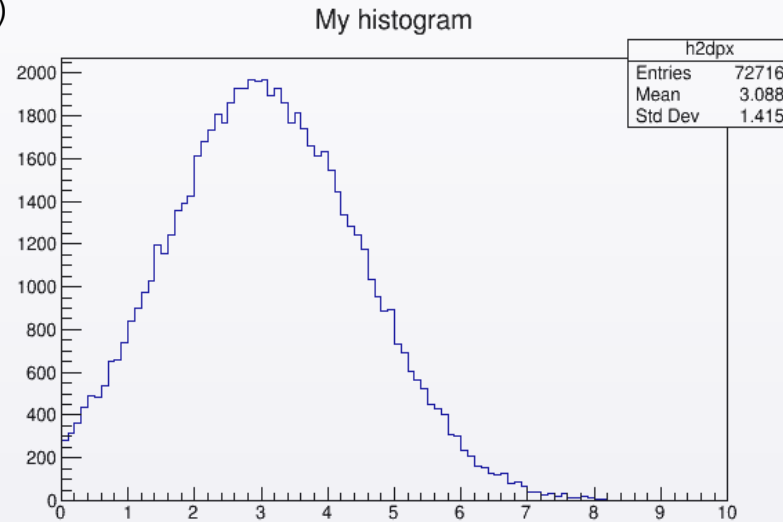
- Access to axis range and bin number (for any axis) – object of **TAxis** class

```
root[5] TAxis* ax = h2d.GetAxis() , * ay = h2d.GetAxis();
root[6] cout << ax->GetXmin() << ' ' << ax->GetXmax() << ' ' << ax->GetNbins() << endl;
root[7] cout << ay->GetXmin() << ' ' << ay->GetXmax() << ' ' << ay->GetNbins() << endl;
root[8] cout << ax->GetBinWidth(1) << ' ' << ay->GetBinWidth(1) << endl;
```

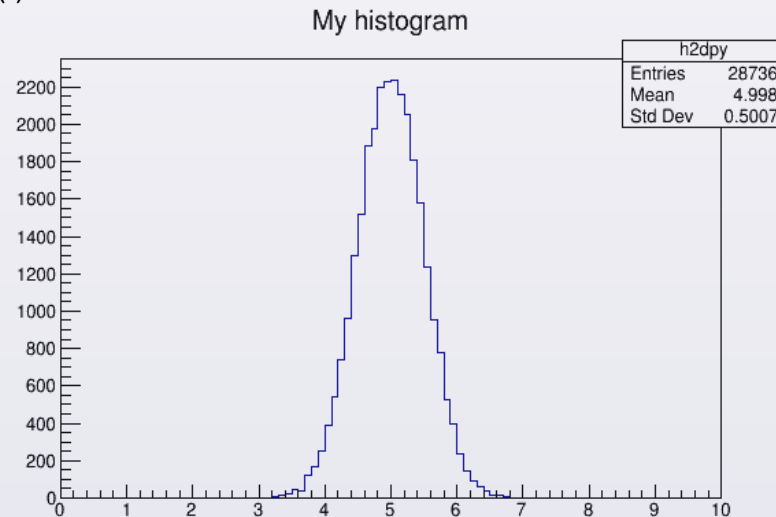
Histograms cont.

► 2dim → 1dim projections (into X axis or Y axis)

```
root[8] h2d.ProjectionX ( "h2dpx", ay->FindBin(4.5) , ay->FindBin(5.5) )
h2dpx(class TH1D *) 0x3660560
root[9] h2dpx->Draw()
```



```
root[10] h2d.ProjectionY ( "h2dpy", ax->FindBin(2.5) , ax->FindBin(3.5) )
root[11] h2dpy->Draw()
```



Basic graphics

- ▶ Points/Markers (`TMarker`), Lines (`TLine`), Arrows (`TArrow`)
- ▶ Boxes (`TBox`), Circles/Ellipses (`TEllipse`)
- ▶ Inscriptions (`TText`), also in the LaTeX style (`TLatex`)

Exemplary help

- Manual: [Graphics chapter](#)
- User guide: [Graphics chapter](#)

Basic objects

[1] **TLine** (Lines)

```
root[1] TH2F h2d ("h2d", "My Histo", 10, 0., 10., 10, 0., 10.); h2d.Draw()
root[2] TLine l1 (0., 0., 1., 1.) ; l1.Draw()
root[3] TLine l2 (0., 0., 1., 1.) ; l2.SetNDC (kTRUE) ; l2.Draw("same")
```

NDC (Normalized Device Coordinates): `SetNDC (0)` ← Coordinates according to actual plot
 `SetNDC (1)` ← as fractions of window dimensions

```
root[4] l2.SetLineColor (2); l2.SetLineStyle (2); l2.SetLineWidth (3)
```



Color numbering scheme in generic palette



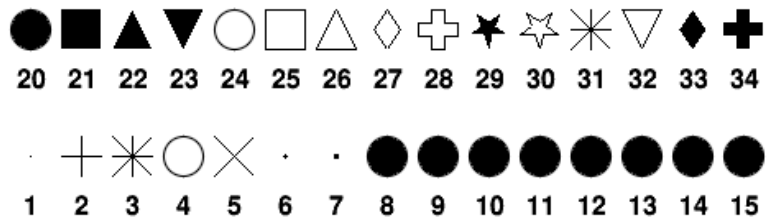
Numbering of line styles

Basic graphics, cont.

Basic objects

[2] **TMarker** (Points / markers)

```
root[1] TMarker m (3., 8., 31)
root[2] m.SetMarkerColor (4);
root[3] m.SetMarkerSize (2.0);
root[4] m.Draw()
```



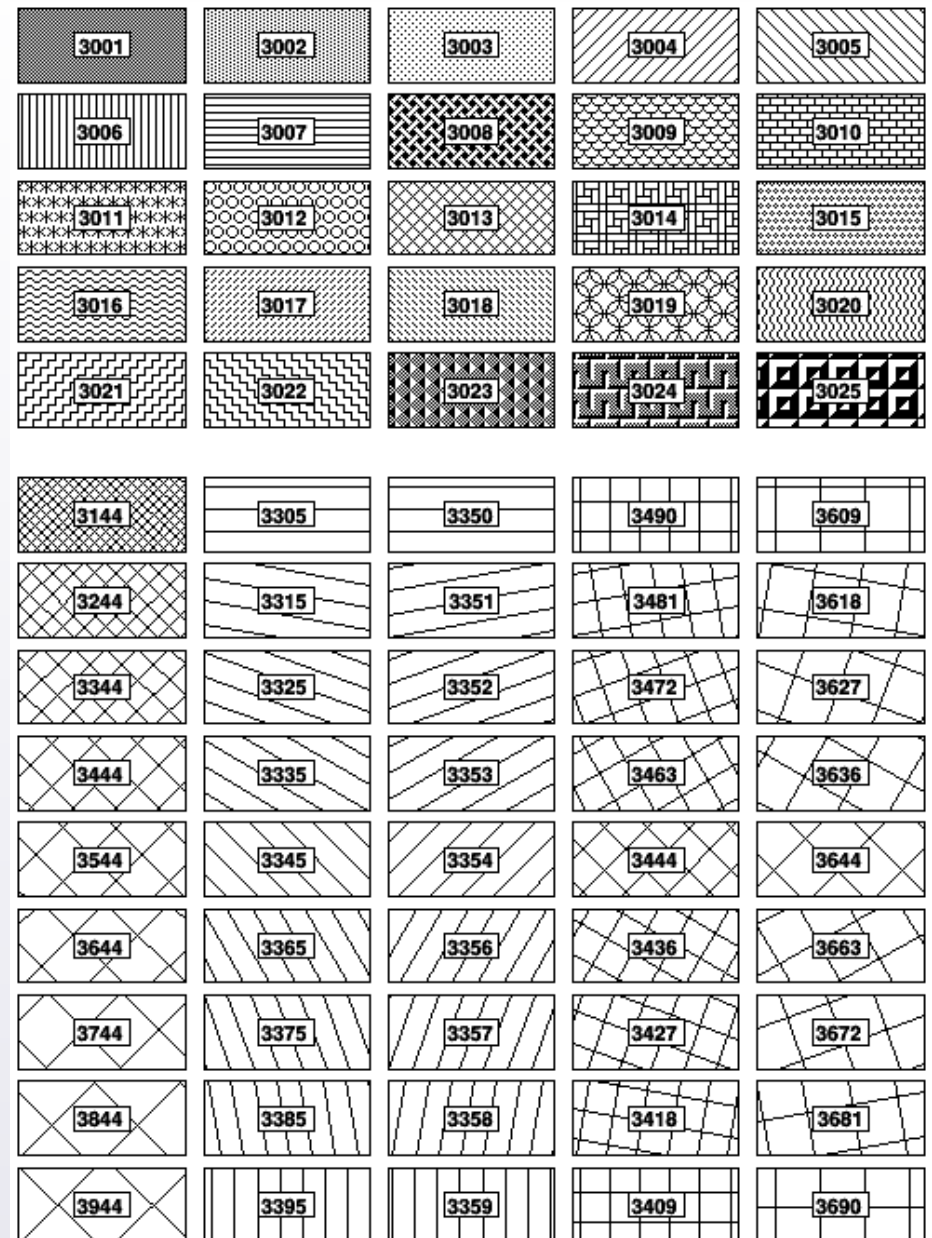
Numbering scheme of markers

[3] **TBox** (Rectangles)

```
root[1] TBox b (5., 2., 8., 3.);
root[2] b.SetLine.....
root[3] b.SetFillColor (4)
root[4] b.SetFillStyle (3014);
```

Conventions of defining the filling style:

root.cern/doc/master/classTAttFill.html#ATTFILL2



Numbering scheme of some hatch styles

Basic graphics, cont.

[4] **TEllipse** (Circles / ellipses)

```
root[1] TEllipse e (  
TEllipse TEllipse(Double_t x1, Double_t y1,  
Double_t r1, Double_t r2 = 0,  
Double_t phimin = 0, Double_t phimax = 360,  
Double_t theta = 0)
```

Range of angles of ellipse fragment : [PhiMin, PhiMax]
Angle of figure rotation : Theta

```
root[1] TEllipse e (5, 5, 4, 2, 0, 270, 45)
```

Line and filling -specific attributes work here.

[5] **TText** (Basic text)

```
root[1] TText t (0.5, 0.5, "Hello World !");  
root[2] t.SetTextColor(2)  
root[3] t.SetTextFont(43)  
root[4] t.SetTextSize(40)  
root[5] t.SetTextAngle(45)  
root[6] t.Draw()
```

▶ Numbering scheme of font styles
Units digit defines the degree of precision

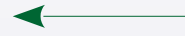
12 : *ABCDEFGH abcdefgh 0123456789 @#\$*
22 : **ABCDEFGH abcdefgh 0123456789 @#\$**
32 : ***ABCDEFGH abcdefgh 0123456789 @#\$***
42 : *ABCDEFGH abcdefgh 0123456789 @#\$*
52 : *ABCDEFGH abcdefgh 0123456789 @#\$*
62 : **ABCDEFGH abcdefgh 0123456789 @#\$**
72 : ***ABCDEFGH abcdefgh 0123456789 @#\$***
82 : *ABCDEFGH abcdefgh 0123456789 @#\$*
92 : *ABCDEFGH abcdefgh 0123456789 @#\$*
102 : **ABCDEFGH abcdefgh 0123456789 @#\$**
112 : ***ABCDEFGH abcdefgh 0123456789 @#\$***
122 : *ABXΔEΦΓH αβχδεφγη 0123456789 ≡#Ξ*
132 : *ABCDEFGH abcdefgh 0123456789 @#\$*
142 : *👉👈👉👈 ℔ ⚡👉👈👉👈 📁📁📁📁 📁📁📁📁*
152 : *ABXΔEΦΓH αβχδεφγη 0123456789 ≡#Ξ*

Basic graphics, cont.

[6] **T_{Latex}** (advanced text: mathematical expressions, etc.)

Helpful summary: root.cern/doc/master/classTLatex.html

```
root[1] TLatex l;  
root[2] l.SetNDC (1);  
root[3] l.SetTextSize (0.06);  
root[4] l.SetTextAngle (45.);  
root[5] l.SetTextColor (4);  
root[6] l.DrawLatex (0.5, 0.6, "E^{2} = m^{2} + p^{2}")
```



A single object acts as a text processor



A few basic rules:

$\wedge\{\dots\}$	: top index	<code>#frac{Numerator}{Denom.}</code>	: Horizontal fraction
$_ \{\dots\}$	: bottom index	<code>#sqrt{x}</code> , <code>#sqrt{N}{x}</code>	: root of degree 2 and higher
<code>#bf{...}</code>	: bold font	<code>#splitline{Above}{Below}</code>	: two lines, one above the other
<code>#it{...}</code>	: italic font	<code>#color[4]{Blue}</code>	: local change of color
<code>#vec{...}</code>	: vector	<code>#font[12]{Font}</code>	: local change of font size
<code>#() {...}</code>	: large brackets	<code>#scale[1.2]{Larger}</code>	: local rescaling of font size

```
root[7] l.DrawLatex (0.5,0.6,"#gamma_{cm} = #frac{1}{#sqrt{1-#beta^{2}}_{cm}}")
```



Some examples from the ROOT site:

$\{ \}^{40}_{20}\text{Ca} :$

$^{40}_{20}\text{Ca}$

$x = \frac{y+z/2}{y^2+1} :$

$x = \frac{y+z/2}{y^2+1}$

Basic graphics, cont.

[6] **TLatex** class, cont.

Codes for Greek symbols (preceded by #)

Lower case		Upper case		Variations
alpha :	α	Alpha :	A	
beta :	β	Beta :	B	
gamma :	γ	Gamma :	Γ	
delta :	δ	Delta :	Δ	
epsilon :	ϵ	Epsilon :	E	varepsilon : ε
zeta :	ζ	Zeta :	Z	
eta :	η	Eta :	H	
theta :	θ	Theta :	Θ	varthetaeta : ϑ
iota :	ι	Iota :	I	
kappa :	κ	Kappa :	K	
lambda :	λ	Lambda :	Λ	
mu :	μ	Mu :	M	
nu :	ν	Nu :	N	
xi :	ξ	Xi :	Ξ	
omicron :	$\omicron$	Omicron :	O	
pi :	π	Pi :	Π	
rho :	ρ	Rho :	P	
sigma :	σ	Sigma :	Σ	varsigma : ς
tau :	τ	Tau :	T	
upsilon :	υ	Upsilon :	Υ	varUpsilonpsilon : Υ
phi :	ϕ	Phi :	Φ	varphi : φ
chi :	χ	Chi :	X	
psi :	ψ	Psi :	Ψ	
omega :	ω	Omega :	Ω	varomega : ϖ

Upper diacritic signs:

#tilde :	$\tilde{a}$
#ddot :	$\ddot{a}$
#dot :	$\dot{a}$
#grave :	$\grave{a}$
#acute :	$\acute{a}$
#check :	$\check{a}$
#hat :	$\hat{a}$
#bar{a} :	$\bar{a}$
#vec{a} :	$\vec{a}$

Basic graphics, cont.

[6] **TLatex** class, cont.

Mathematical and other special symbols

$\clubsuit$	#club	$\Leftrightarrow$	#Leftrightarrow	$\leftarrow$	#leftarrow	∇	#nabla	$\grave{a}$	#aa
$\wp$	#voidn		#void8	$\otimes$	#otimes	$\swarrow$	#downleftarrow	/	#/
$\leq$	#leq	$\hbar$	#hbar	$\Leftarrow$	#Leftarrow	$\top$	#topbar	\	#backslash
$\approx$	#approx	$\blacklozenge$	#diamond	$\prod$	#prod	$\overbrace{}$	#arcbar	$\cdot$	#upoint
$\in$	#in	$\aleph$	#aleph	$\square$	#Box	$\uparrow$	#uparrow	∂	#partial
$\supset$	#supset	$\geq$	#geq	$\parallel$	#parallel	$\oplus$	#oplus	$\lrcorner$	#corner
$\cap$	#cap	$\neq$	#neq	$\heartsuit$	#heart	$\Uparrow$	#Uparrow	$\{$	#lbar
$\copyright$	#copyright	$\notin$	#notin	$\mathfrak{S}$	#Jgothic	$\sum$	#sum	$\lfloor$	#bottombar
TM	#trademark	$\subseteq$	#subseteq	$\langle$	#LT	$\perp$	#perp	$\rightarrow$	#rightarrow
$\times$	#times	$\cup$	#cup	$\equiv$	#equiv	$\forall$	#forall	$\surd$	#surd
$\bullet$	#bullet	$\copyright$	#copyright	$\subset$	#subset	$\spadesuit$	#spade	$\Rightarrow$	#Rightarrow
f	#voidb	TM	#void3	$\supseteq$	#supseteq	$\mathfrak{R}$	#Rgothic	$\int$	#int
$"$	#doublequote	$\div$	#divide	$\wedge$	#wedge	$\rangle$	#GT	$\odot$	#odot
	#lbar	$\circ$	#circ	$\circledcirc$	#oright	$\propto$	#propto	$\exists$	#exists
$\frown$	#arcbottom	∞	#infty	$\text{\AA}$	#AA	$\not\subset$	#notsubset	$+$	#plus
$\downarrow$	#downarrow	$\angle$	#angle	$\pm$	#pm	$\oslash$	#oslash	$-$	#minus
$\leftrightarrow$	#leftrightarrow		#cbar	$\mp$	#mp	$\vee$	#vee		
$\Downarrow$	#Downarrow	(	#arctop	$\cdots$	#3dots	$\circledcirc$	#void1		

TStyle Object managing the graphical style

- ▶ In ROOT session, the **gStyle** object of **TStyle** class is available. It keeps settings of graphical style. The settings concern e.g. the attributes of canvas, lines, markers, stats. Access is through the getters and setters.

```
root[1] gStyle->Set   
  
root[1] gStyle->SetLabelSize (0.07, "XY");  
root[2] gStyle->SetLabelOffset (0.01, "Y" );  
root[3] gStyle->SetNdivisions ( 2 , "X" );  
root[4] TH1F h1 ("h1", "", 10, -5, 5);  
root[5] h1.Draw ();
```

- ▶ However, if we first defined a histogram, and changed the style later on, we need to inform that histogram in order it to update the style:

```
root[6] gStyle->SetNdivisions ( 8 , "X" );  
root[8] h1.Draw ();  
root[9] h1.UseCurrentStyle ();  
root[10] h1.Draw ();
```

- ▶ Important: TStyle allows to modify the contents of displayed statistics:

```
root[11] gStyle->SetOptStat ( {rmen} );
```

Symbols {r m e n} are the basic 4 of 9 attributes to display. They can take values {0, 1}, sometimes 2.

Basic properties:

r	=	(0) 1:	(do not) display RMS
m	=	(0) 1:	(do not) display the mean
e	=	(0) 1:	(do not) display the count numbers
n	=	(0) 1:	(do not) display the histogram name

[Link to the full list.](#)

TFile **Root** files for object storage

(root.cern/root/html/doc/guides/users-guide/InOutOutput.html
root.cern/doc/master/classTFile.html)

To begin with:

```
root[1] TFile* f = new TFile ("myfile.root", "RECREATE");
root[2] TH1F *h = new TH1F ("myhisto", "My Histo's Title", 10, -5., 5.);
root[3] TRandom3 r;  for (int i=0; i<1e5; i++) h->Fill ( r.Gaus() );
root[4] h->Write ();                                     ← Writing object to file
root[5] f->Close ();
root[6] .! ls -l myfile.root

root[1] TFile* f = new TFile ("myfile.root", "READ");   ← Opening file for reading
root[2] TH1F* hread = (TH1F*) f->Get ("myhisto");
root[3] hread->Draw ();
root[4] f->Close ();
```

A closer look:

```
root[1] TFile* f = new TFile ("myfile.root", option);

                                     option = "NEW" "RECREATE" "UPDATE" "READ"

root[2] if (f->IsOpen () == true) cout << "File open.\n";
root[2] f->ls();
root[3] TH1F* h = new TH1F ("h", "myhisto", 10, 0., 10.);
root[4] f->ls();                                     ← h linked, but not yet saved.
root[5] .! ls -l myfile.root
root[6] h->Write ();                                   ← h linked and saved.
root[7] f->ls();
root[8] .! ls -l myfile.root
root[9] h->Write ("h_copy");                           ← h written under the new name
root[10] f->ls ();
root[11] f->Close();
```

I/O cont.



ROOT session with file connection:

```
> nice root -l myfile.root
root[0]
Attaching file myfile.root as _file0...
(class TFile *) 0x1943c70
root[1] _file0->ls();
```



Pointer to object of TFile class



Setting up the work directory on a disk:

```
root[2] gSystem->pwd ()
(const char *) "/home/krzysztof/didact/informatyka/nuctools"
root[3] gSystem->cd ("../")
root[4] gSystem->pwd ()
(const char *) "/home/krzysztof/didact/informatyka/"
```



Execution of Linux command :

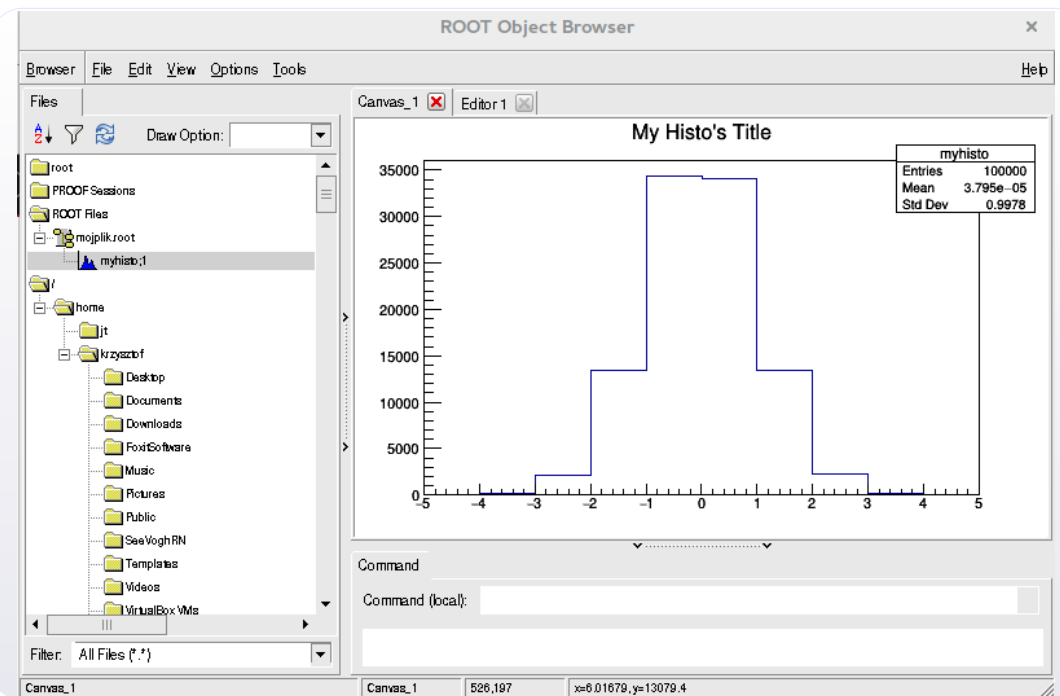
```
root[4] gSystem->Exec ("date")
Tue, Nov 5, 09:55:01 CET

root[5] TString datenow =
gSystem->GetFromPipe ("date")
```



TBrowser : the object browser

```
root[5] new TBrowser
```



gDirectory Hierarchy of objects in Root files and memory

► ROOT maintains a **structure of directories** (in memory, as well as inside .root files) .

```
> nice root -l
root[0] gDirectory->pwd ()          or          .pwd
Current directory: Rint:/
                                     ← Main directory in memory

root[1] TFile f1 ("myfile.root"); cout << gDirectory->GetPath () << endl;
Myfile.root:/

root[2] gDirectory->ls ()          or          .ls
TFile**          myfile.root
TFile*           myfile.root
KEY: TH1F        h;1          myhisto
                                     ← We see we've moved to a file
```

► **Creating subdirectories** (in memory or inside a .root file)

```
root[3] TFile f2 ("newfile.root", "RECREATE")
root[4] gDirectory->mkdir ("folder1");
root[5] gDirectory->cd ("folder1"); gDirectory->pwd ()
newfile.root:/folder1
                                     ← New directory in file

root[6] TH1F h ("myhisto", "", 10, -5., 5); h.Write();
root[7] .ls
TDirectoryFile*          folder1          folder1
OBJ: TH1F                myhisto : 0 at: 0x7f06ee3ce000
KEY: TH1F                myhisto;1
                                     ← myhisto saved in folder1

root[8] gDirectory->cd ("..") ; gDirectory->rmdir ("folder1")
root[8] f2.Close ();
root[9] cout << gDirectory->GetPath() << endl;
Rint:/
Root[10] gDirectory->cd ("Rint:/"); gDirectory->pwd(); ← Return to main directory
```

MACROS : C++/ROOT codes in a file

Handy mode: `macro_noname.C`

```
{
  TH1F h1 ("hist1", "", 50, -5., 5.);
  TH1F* h2 = new TH1F
    ("hist2", "", 50, -5., 5.);

  TRandom3 r;  r.SetSeed ();

  for (int i=0 ; i<1e5 ; i++) {
    h1.Fill ( r.Gaus() );
    h2->Fill ( r.Gaus() );
  }
  h1.Draw();
}
```

After execution, in an interactive session:

- Existent: h1 object and *h2 pointer
- One can call: h1, h2, hist1, hist2

Full mode (functions):

Name the same
as filename

```
int macro_function () {
  TH1F h1 ("hist1", "", 50, -5., 5.);
  TH1F* h2 = new TH1F
    ("hist2", "", 50, -5., 5.);

  TRandom3 r;  r.SetSeed ();

  for (int i=0 ; i<1e5 ; i++) {
    h1.Fill ( r.Gaus() );
    h2->Fill ( r.Gaus() );
  }

  //TCanvas can1 ("c1", "", 640, 480);

  h1.Draw();

  //can1.Update();
  //cin.ignore();

  return 0;
}
```

After execution, in an interactive session:

- h1 object not present (also via hist1)
- One cannot call h2
- POSSIBLE to call hist2

Additional commands needed to capture graphics

MACROS cont.

Input arguments of function:

```
double macro_inputarg (double x = 0)
{
    cout << "Hello world! " << x << endl;
    return x;
}
```

```
> nice root -b "macro_inputarg.C(12.34) "
```

← No space between .C and (...)

Calling from session:

```
root[0] .x macro_inputarg.C (-12.)
```

Macro can contain many functions.
The starting function (equivalent of `main`)
MUST have the same name as filename



```
Double_t W (Double_t x) {
    return 3*pow(x,2) - 1.5*x + 4.;
}

int macro_giveW (double x) {
    cout << "W(x) = " << W(x) << endl;
    return 0;
}
```

**A macro can be loaded first
and executed later.**

```
root[0] .L macro_inputarg.C
root[1] macro_inputarg (123) ;
Hello world! 123
```

← This way you can quickly check the integrity of C++ code.

One can compile the macro in the session. But the code must contain `#include< ... >`

```
root[0] .L macro_inputarg.C+
root[1] macro_inputarg (-123.45);
```

← Will create file `macro_inputarg_C.so`

TLegend Legend for a plot

► Each plot can be accompanied with a legend (one or more).

An object of **TLegend** class needs to be linked with the plotted: functions, histograms or graphs.
We decide, what inscription and symbol we assign (line, marker, rectangle, point with uncertainty).

```
int macro_TLegend () {
    TH1F* h = new TH1F ("h", "Example", 200, -14, 10);
    h->FillRandom ("gaus", 30000);
    h->SetFillColor (18);
    h->Draw ();

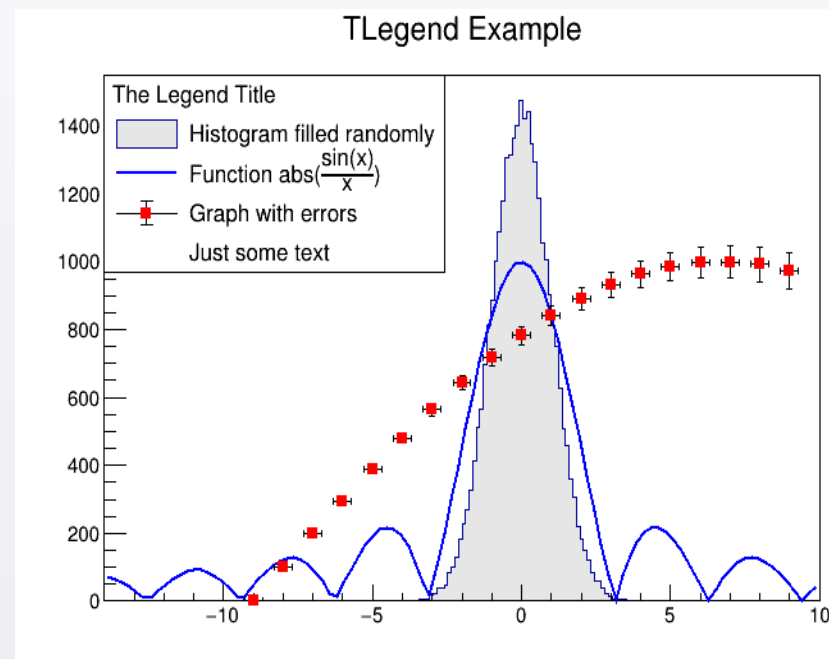
    TF1* f= new TF1("f", "1000*abs(sin(x)/x)",-14,14);
    f->SetLineColor (kBlue);
    f->Draw ("same");

    Int_t i = 0;
    Double_t x[50], y[50], ex[50], ey[50];
    for (Double_t xval = -9; xval <= 9; xval++, i++){
        x[i] = xval;
        y[i] = 1000 * sin ( (xval + 9)/10 ) ;
        ex[i] = 0.3;
        ey[i] = (xval+9) * 3 ;
    }
    TGraphErrors* gr = new TGraphErrors (i,x,y,ex,ey);
    gr->SetName ("gr");
    gr->SetMarkerStyle (21);
    gr->SetMarkerColor (kRed);
    gr->Draw ("P");

    TLegend* leg = new TLegend (0.1, 0.6, 0.48, 0.9);
    leg->SetHeader ("The Legend Title");
    leg->AddEntry ( h ,"Histogram filled rand", "f");
    leg->AddEntry ("f" ,"abs(#frac{sin(x)}{x})", "l");
    leg->AddEntry ("gr","Graph with errors" , "lep");
    leg->AddEntry ((TObject*)0, "Just some text", "");
    leg->Draw ();
    return 0;
}
```

• Legend must be displayed by command

This macro draws histogram, function and TGraph.
Next, it creates the legend for them
And modifies the symbols' attributes in the legend.



← Creating legend within certain canvas area
← Legend's title

Entries corresponding to objects.

Options: f = filled box, l = line

p = marker e = error bar

• In case if text only

Good plot in ROOT

► This macro is a proposition of way of creating a readable and clear plot.

```
int macro_GoodPlotExample () {  
  gStyle->SetOptStat (0);  
  gStyle->SetLegendBorderSize (0);  
  gStyle->SetLegendTextSize (0.055);  
  gStyle->SetLabelSize (0.055, "XY");  
  gStyle->SetNdivisions ( 505 , "XY");  
  
  {... creation of histogram, curve and TGraphErrors ...}  
  
  TCanvas* c1 = new TCanvas ("c1", "", 800, 600);  
  c1->SetGrid (0, 0);  
  c1->SetTopMargin (0.05);  
  c1->SetBottomMargin (0.15);  
  c1->SetRightMargin (0.04);  
  c1->SetLeftMargin (0.16);  
  
  h1->Draw ();  
  f1->Draw ("same");  
  gr->Draw ("P");  
  
  TLegend* leg = new TLegend  
    (0.21, 0.63, 0.5, 0.86, "", "nbNDC");  
  leg->AddEntry ( h1 , " Experiment" , "f" );  
  leg->AddEntry ("f1", " Model" , "l" );  
  leg->AddEntry ("gr", " Efficiency" , "lep");  
  leg->Draw ();  
  
  TLatex l;  
  l.SetNDC (1);  
  l.SetTextSize (0.055);  
  l.DrawLatex (0.44, 0.025, "Position (#mum)");  
  l.SetTextAngle (90);  
  l.DrawLatex (0.035, 0.34 , "Rate (arb. units)");  
  l.SetTextAngle (0);  
  return 0;  
}
```

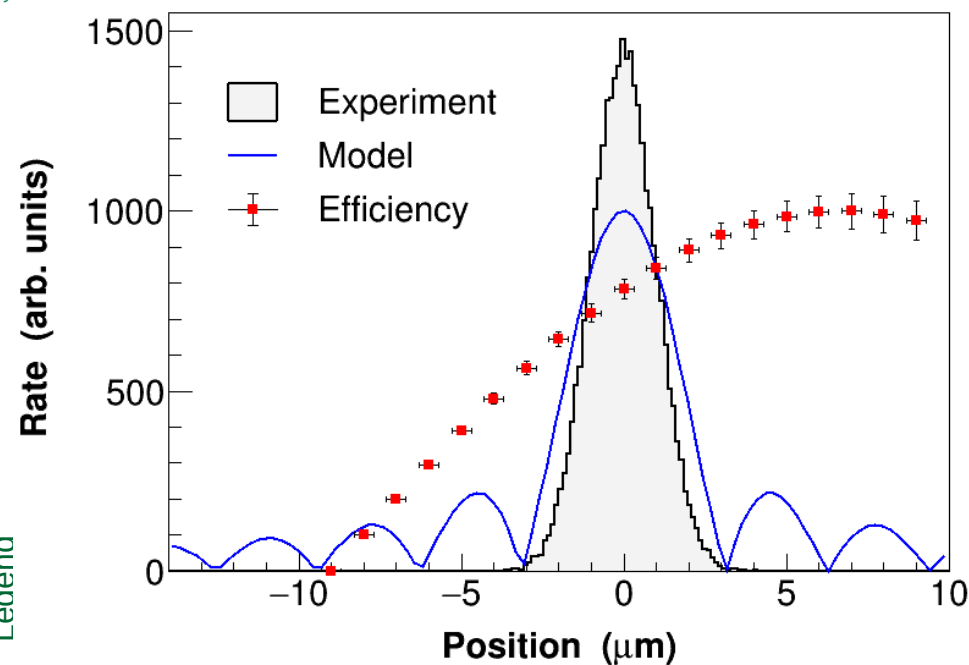
Style settings

Do not plot stats
Legend: no frame
Legend: font size
Font size of axis values
Number of divisions of X and Y axis

Margins

Legend

Axis labels



Use coordinates of TCanvas
Font size
Print the X axis label
Change the angle to vertical printing
Print the Y axis label

MACROS : C++ functions used by TFn objects

Through “Inline Expression” :

```
Double_t W (Double_t x) {  
    return pow(x,3) - 6.*pow(x,2)  
        + x - 1.;  
}  
  
int macro_funcInline ()  
{  
    TCanvas c1 ("c1", "", 640, 480);  
    TF1 f1 ("f1", "W(x)", -10., 10.);  
    f1.Draw();  
    c1.Update();  
    cin.ignore();  
    return 0;  
}
```

Through function with parameters:

```
Double_t myFun (Double_t *xarg,  
                Double_t *par)  
{  
    Double_t x = xarg[0] , result = 0.;  
  
    for (int deg = 0 ; deg <= 3 ; deg++)  
        result += par[deg]  
            * TMath::Power (x, deg);  
    return result;  
}  
  
int macro_funcFunc ()  
{  
    TCanvas c1 ("c1", "", 640, 480);  
    TF1 f1 ("myfun1", myFun, -3, 5, 4);  
    f1.SetParameters (-1., 1., -6., 1.);  
    f1.Draw();  
    c1.Update();  
    cin.ignore();  
    return 0;  
}
```

Notice:

in myFun we can encode anything
e.g. if/else blocks or calls to other functions

Fitting of function to data points

Please download the test data: `wget www.fuw.edu.pl/~kpias/ctnp/dataPoints.txt`

```
void macro_FitTGraphErrors () {  
    TCanvas* c1 = new TCanvas ("c1");  
    TGraphErrors gr ("dataPoints.txt");  
    gr.SetTitle ();  
    gr.Draw("AP");  
  
    TF1 fun ("fun", myFun , -3, 5, 4);  
    fun.SetParameters (-1. , 1. , -6. , 1.);  
    gr.Fit ( &fun );  
  
    c1->Update();  
    cin.ignore();  
}
```

Graph with
experimental data

Model function
(with parameters - !)

Fit command.
MINUIT package
(from CERN)

Exemplary result of (successful) fit:

χ^2

Status of fit (✓)

FCN=10.8023 FROM MIGRAD		STATUS=CONVERGED		91 CALLS	92 TOTAL
		EDM=5.47088e-07		STRATEGY= 1	ERROR MATRIX ACCURATE
EXT NO.	PARAMETER NAME	VALUE	ERROR	STEP SIZE	FIRST DERIVATIVE
1	p0	-1.31434e+00	3.32403e-01	3.54860e-04	1.79927e-03
2	p1	1.25659e+00	3.08140e-01	2.52787e-04	4.11222e-03
3	p2	-5.75515e+00	1.76173e-01	1.13715e-04	1.26442e-02
4	p3	9.12643e-01	7.84954e-02	4.27100e-05	3.82217e-02

Status of
covariance matrix
(✓)

Found values
of parameters

Values

Uncertainties

Fitting of function to data points cont.

▶ Setting up the initial parameters:

```
fun1.SetParameter (index, value) ;  
fun1.SetParameters (index, index, ... , index);  
fun1.FixParameter (index, value) ;  
  
fun1.SetParLimits (index, min, max) ;  
gr.Fit ( &fun , "" );
```

- ← Set parameter's value
- ← Set all the parameters
- ← Fix a given parameter
- ← Set the fitting range

▶ Getting the values of found parameters:

```
fun1.GetParameter (index) ;  
fun1.GetParError (index) ;  
fun1.GetChisquare ();  
fun1.GetNDF ();
```

- ← Get parameter's value
- ← Get parameter's error
- ← Get χ^2 value
- ← Get number of d.o.f.

▶ Usage of built-in (predefined) functions :

gr.Fit ("pol3");	polN	:	$f(x) = p_0 + p_1x + p_2x^2 + \dots$
	gaus	:	$f(x) = p_0 \exp(-0.5 * ((x-p_1)/p_2)^2)$
	gausn	:	(Normal Distribution)
	expo	:	$f(x) = \exp(p_0 + p_1x)$
	landau	:	(Landau distribution of energy losses)
	chebyshevN	:	(Chebyshev polynomial of degree N)

Caution: for a predefined function, if we narrow down the range of parameters (or fix some value(s)), we have to add "B" into the option of Fit method.

Fitting of function to data points cont.

▶ The `Fit` method works also for histograms, including 2, 3 – dimensional ones. Full form of method:

```
TFitResultPtr Fit(TF1* f1, Option_t* option = "",           ← Fitting options
                  Option_t* goption = "",                 ← Drawing options
                  Double_t xmin = 0, Double_t xmax = 0)    ← Range on X axis
```

▶ **Fitting options** (selection of more practical ones; for details see [this link](#))

only for histograms (`THdf`) :

I	(Integral)	Average the function over each bin (for strongly changing functions)
L	(LogLikelihood)	Use the Log Likelihood method (instead of χ^2).

For histograms and graphs (`TGraph_____`) :

M	(iMprove)	Obtain the more precise fit results
E	(Error)	Obtain uncertainties more exactly with help of Minuit's MINOS package.
B	(Bound)	For predefined functions: if range of parameter values is limited
R	(Range)	Fit in range, in which the function is defined
0		Do not plot the fitted function
V	(Verbose)	Verbose mode
Q	(Quiet)	Quiet mode

▶ In case of fitting the 2 (3) – dimensional function to the 2 (3) – dimensional function:

- The fitting range should be specified in the constructor of the `TFn` object
- Add "R" to the fitting options of the `Fit` method.
- If you specify `xmin` and/or `xmax` in arguments of `Fit` method, these values work only for X axis

Fitting of function to data points **cont.**

► Extraction of **covariance matrix** (goal: linear correlations between fitted parameters)

```
void macro_FitErrorMatrix () {  
    TCanvas *c1 = new TCanvas ("c1");  
    TGraphErrors gr ("dataPoints.txt");  
    gr.SetTitle ();  
    gr.Draw("AP");  
  
    TF1 fun ("fun", myFun , -3, 5, 4);  
    fun.SetParameters (-1. , 1. , -6. , 1.);
```

```
    TFitResultPtr fitRes = gr.Fit ( &fun , "S" );  
    TMatrixDSym cov = fitRes->GetCovarianceMatrix();
```

```
    for (int r = 0; r < cov.GetNrows () ; r++) {  
        for (int c = 0; c < cov.GetNcols () ; c++)  
            cout << setw(16) << cov[r][c] ;
```

```
        cout << endl;  
    }
```

```
    c1->Update();  
    cin.ignore();  
}
```

TFitResultPtr
stores the fit results

TMatrixDSym
symmetric matrix of
double elements

GetCovariance...
returns the covariance
matrix

► To access the **fit status** inside the code (string)

```
#include "TMinuit.h"  
string myFitStatus = gMinuit->fCstatu ;
```

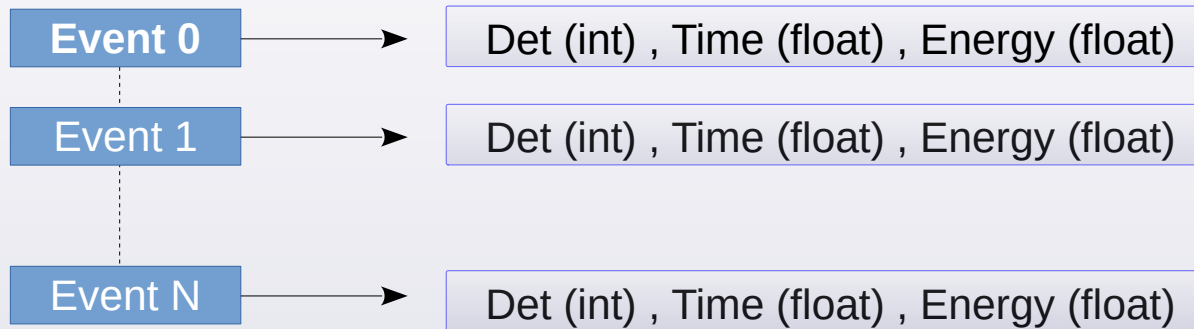
At the beginning of the code
Getting the status (string)

TTree Trees (data bases)

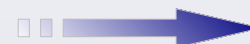
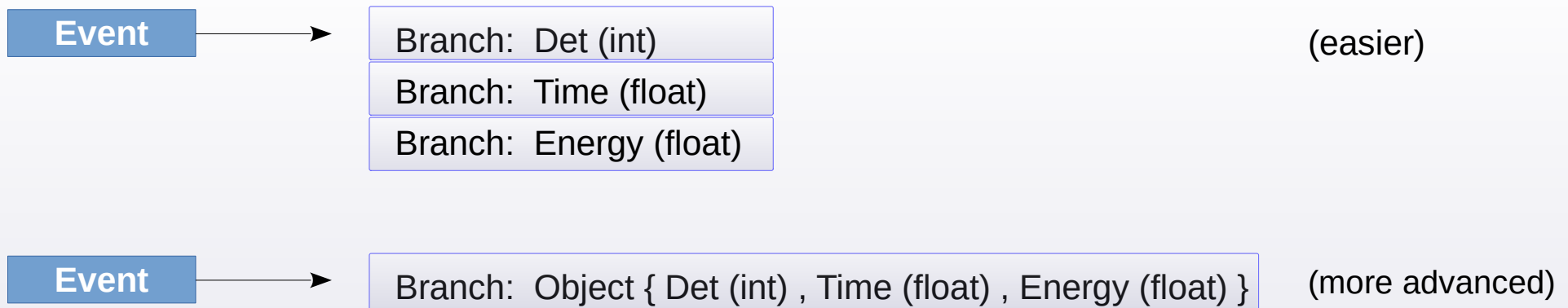
Eg. experiment measuring particles in telescopes: set of N_i , T_i , ΔE_i , E_i from a detector (detectors)

Eg. experiment measuring tracks of particles in drift chambers : set of p_x , p_y , p_z , ΔE_i from a chamber

A simple data scheme



Possible event structures:



Let's first look at variant 1.

TTree Simplest way to create a tree from data in text file

- ① Download data. They are in 3 columns (int, float, float)
wget www.fuw.edu.pl/~kpias/ctnp/MyExpData.txt

11	16.832	10.8703
11	20.4335	8.65938
7	0.634218	8.03354
1	21.3472	19.6014
...		

- ② Let's open the TTree object and fill the database with data using the `ReadFile` method:

Declaration of tree

`ReadFile` fills the tree
from txt file

`Print` displays stats

`Scan` prints out data

`Draw`: en-time plot

We set up filter on `det`

`Write` saves tree in file

```
int TTree_ReadFile ()
{
    TFile f ( "simplest_tree.root", "RECREATE" );
    TTree t ( "mytree" , "Tree of data for my analysis" );
    t.ReadFile ("MyExpData.txt", "det/I:energy/F:time/F" );
    t.Print ();
    t.Scan ( "det:energy:time" );
    t.Draw ( "energy:time" , "det >= 7" );
    t.Write ();
    return 0;
}
```

TTree cont. Now we'll design the tree by ourselves.

► Scheme "1 branch = 1 variable"

Define the branch:

```
t.Branch ("Name" ,  
         &variable,  
         "variable/F");
```

Encoding the variable size:

F	: float	, 4 bytes
D	: double	, 8 bytes
I	: signed integer	, 4 bytes
i	: unsigned integer	, 4 bytes
C	: c-string	
B	: signed integer	, 1 byte
b	: unsigned integer	, 1 byte
S	: signed integer	, 2 bytes
s	: unsigned integer	, 2 bytes
L	: signed integer	, 8 bytes
l	: unsigned integer	, 8 bytes
O	: bool	, 1 bit

```
int TTree_simple () {  
    Int_t det;  
    Float_t energy , time;  
  
    TFile f ("simple.root", "RECREATE");  
    TTree t ("tree", "My tree");  
    t.Branch ("Det" , &det , "det/I");  
    t.Branch ("En" , &energy, "energy/F");  
    t.Branch ("Time", &time , "time/F");  
  
    TRandom3 r; r.SetSeed ();  
    for (int i = 0; i < 100; i++) {  
        det      = r.Integer (24);  
        time     = r.Rndm() * 20.;  
        energy   = r.Rndm() * 30.;  
        t.Fill ();  
    }  
    t.Write ();  
    return 0;  
}
```

Making an entry in the tree:

```
t.Fill ()
```

Writing the tree in a file:

```
t.Write ()
```

TTree cont.

► Inspection of the tree in an interactive session

```
nice root -l simple.root
```

```
root[0] tree->Print ()
```

```
*****
*Tree      :tree      : My tree                                     *
*Entries   :      100 : Total =           3169 bytes  File  Size =       1701 *
*          :          : Tree compression factor =    1.21                *
*****
*Br       0 :Det       : det/I                                       *
*Entries   :      100 : Total  Size=        936 bytes  File Size  =        230 *
*Baskets   :         1 : Basket Size=    32000 bytes  Compression=    2.04    *
*.....*
*Br       1 :En        : energy/F                                     *
*Entries   :      100 : Total  Size=        943 bytes  File Size  =        469 *
*Baskets   :         1 : Basket Size=    32000 bytes  Compression=    1.00    *
*.....*
*Br       2 :Time      : time/F                                       *
*Entries   :      100 : Total  Size=        941 bytes  File Size  =        471 *
*Baskets   :         1 : Basket Size=    32000 bytes  Compression=    1.00    *
*.....*
```

```
root[1] tree->Show (10)
```

```
=====> EVENT:10
```

```
det          = 10
energy       = 3.10897
Time        = 5.81155
```

```
root[2] tree->Scan ()
```

```
*****
*      Row      * Det.Det.d * En.En.ene * Time.Time *
*****
*          0 *          1 * 2.7607548 * 2.8281364 *
*          1 *          12 * 13.696406 * 2.2420666 *
*          2 *          12 * 21.884300 * 11.228475 *
*          3 *          11 * 10.673481 * 10.060612 *
*          4 *          17 * 16.964376 * 18.435609 *
*          5 *           5 * 9.2536840 * 7.2596163 *
```

TTree cont.

▶ Plotting the histogram of a variable (variables, combination of variables, etc)

```
root[0] tree->Draw ("energy")
```

```
root[1] tree->Draw ("sqrt(energy)")
```

← Example of function of variable

```
root[2] tree->Draw ("time:energy", "", "colz")
```

← 2-dimensional plot

```
root[3] tree->Draw ("time:Entry$" )
```

← Entry\$ is a special keyword
= entry number

▶ Plotting the histogram of a variable with some filters (cuts) required

```
root[4] tree->Draw ("time", "det>14 && det<23")
```

▶ Projection of variables from a tree to a histogram

```
root[5] tree->Project ( 
Long64_t Project(const char* hname, const char* varexp, const char* selection
= "", Option_t* option = "", Long64_t nentries = 1000000000, Long64_t
firstentry = 0)
```

```
root[5] TH1F henergy ("henergy", "", 15, 0., 30. );
```

```
root[6] tree->Project ("henergy", "energy", "det<=10" );
```

```
root[7] henergy.Draw ();
```

▶ Cuts (TCut)

```
root[8] TCut cut1 ("det<=10") , cut2 = "det>=20" ;
```

```
root[9] henergy.Reset ();
```

```
root[10] tree->Project ("henergy", "energy", cut1 || cut2 );
```

```
root[11] tree->Draw ("energy", cut1 && "Entry$ <= 50" );
```

← One can combine
TCut with *string*

TTree cont.

► Graphical cut performed on TH2 (TCutG)

```
$ wget www.fuw.edu.pl/~kpias/ctnp/auau_1.23AGeV_8dsts.root
$ nice root -l auau_1.23AGeV_8dsts.root
```

```
root[1] wars_tree->Draw ("dEdxToF:totmom", "totmom<1500 && dEdxToF<15.");
```

- ① From the menu of the TCanvas: **View→Toolbar**, next .
- ② By clicking mouse – mark the vertices of the polygon. Double click to finish.
- ③ A pointer to the TCutG object is available in the session: TCutG* CUTG.

```
root[2] wars_tree->Draw ("dEdxToF:totmom", "totmom<1500 && dEdxToF<15. && CUTG")
root[3] CUTG->Draw ("same")
```

By using the `IsInside` method we can examine if the pair of coordinates lies inside the contour, e.g.:

```
root[4] CUTG->IsInside ( 500, 7 );
```

The cut object can be renamed, as well as stored in a root file:

```
root[5] CUTG->SetName ( "mycutg" );
root[6] TFile f ( "mycutg.root" , "recreate"); mycutg->Write();
```

However, if we want to get it from a file and use as a tree selection, we first have to assign variables to axes:

```
$ nice root -l auau_1.23AGeV_8dsts.root
root[0] TFile filecut ( "mycutg.root" );
root[1] TCutG* cg1 = (TCutG*) filecut.Get ("mycutg");
root[2] cg1->SetVarX ("totmom"); cg1->SetVarY ("dEdxToF");
root[3] _file0->cd();
root[4] wars_tree->Draw ("dEdxToF:totmom", "totmom<1500 && dEdxToF<15. && mycutg");
```

► Getting the TTree from the ROOT file + readout of data from TTree:

① Connect to the tree:

```
TTree* t = (TTree*) f.Get ("tree");
```

② Connect the variables to the branches:

```
t->SetBranchAddress ("name",&variable);
```

③ Get the number of entries:

```
t->GetEntries();
```

④ Read the full event into the variables:

```
t->GetEntry (i);
```

```
int TTree_simple_read () {  
    Int_t det;  
    Float_t energy , time;  
  
    TFile f ("simple.root");  
    TTree* t = (TTree*) f.Get ("tree");  
  
    t->SetBranchAddress ("Det" , &det );  
    t->SetBranchAddress ("En" , &energy);  
    t->SetBranchAddress ("Time", &time );  
  
    for (int i=0; i<t->GetEntries(); i++)  
    {  
        t->GetEntry (i);  
        cout << setw( 5) << det  
              << setw(12) << time  
              << setw(12) << energy << endl;  
    }  
    return 0;  
}
```

► Status of branches

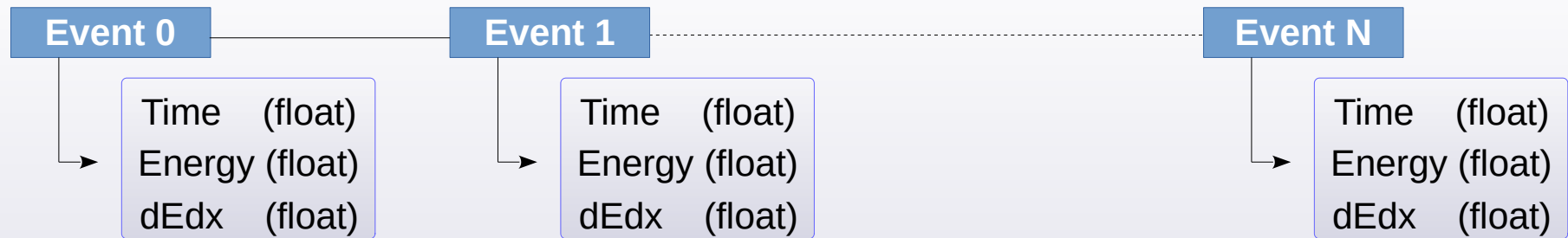
One can define, which branches should be analysed, and which ones – not. Before the event loop, one should issue:

```
TTree::SetBranchStatus ("branch", status);  status:  (0) / 1 = (in)active  
                                                "branch" = "*" : (de)activation concerns all the branches
```

Notice: Deactivation of unnecessary branches **shortens** the analysis time (important for large data files!)

TNtuple (TNtupleD) Simple trees composed of only *floats* (*doubles*)

Even simpler data scheme (case of floats) :



- ▷ Variables are defined in the constructor.
- ▷ For every variable a branch is made.
- ▷ Filling is done by giving variable values
- ▷ Direct filling up to 15 variables:

```
Fill (var1, var2, ...)
```

- ▷ ... or via array:

```
Fill (Float_t* x)
```

```
int TNtuple_example () {  
    Float_t energy , time, dEdx;  
  
    TFile f ("tntuple.root", "RECREATE");  
    TNtuple n ("tntuple", "My ntuple", "Time:En:dEdx" );  
  
    TRandom3 r; r.SetSeed ();  
    for (int i=0; i<100; i++) {  
        time    = r.Rndm() * 20.;  
        energy   = r.Rndm() * 30.;  
        dEdx     = r.Rndm() * 0.5;  
        n.Fill (time, energy, dEdx);  
    }  
    n.Write();  
    return 0;  
}
```

- **Merging data from ROOT files with the same structure**

If we need to analyse a series of files with TTree that has the same structure, we can of course make a loop: open i-th file, connect the tree and branches, analyse data, and close that file. However, if we store the resulting histograms in a common output file, one often has to switch back and forth the `gDirectory`.

There is an alternative: [merging the input data](#).

▶ **TChain**. Object being effectively a queue of subsequent TTrees in specified files. Let's assume that every input file has a TTree called "T".

- ① Create the TChain:

```
TChain myChain ("T");
```
- ② Add subsequent files:

```
myChain.Add ("file1.root");  
myChain.Add ("file2.root");  
myChain.Add ("file3.root");
```
- ③ Since now we use the `myChain` object, as if it was the common input tree.

▶ The **hadd** executable, runnable from prompt :

```
> hadd data_merged.root data_1.root data_2.root ....  
      (or: data_*.root)
```

Caution: the maximum size of resulting file is set to **100 GB**.

For bigger data there is a **TFileMerger** class. One can use [this macro](#).

TTree cont. Handling the TVectorN {N = 2, 3} / TLorentzVector object in an event:

► Storage:

```
int TTree_TVector () {  
  
    TVector3 v3;  
    TVector3* pv3 = &v3;  
  
    TLorentzVector vL;  
    TLorentzVector* pvL = &vL;  
  
    TFile file ("TTree_TVector.root", "recreate");  
  
    TTree* ttree = new TTree ("ttree", "ttree");  
    ttree->Branch ("v3", "TVector3", &pv3);  
    ttree->Branch ("vL", "TLorentzVector", &pvL);  
  
    TRandom3 r; r.SetSeed (0);  
  
    for (int evt = 0; evt < 100; evt++)  
    {  
        v3.SetXYZ (r.Rndm(), r.Rndm(), r.Rndm());  
        vL.SetXYZT (r.Rndm(), r.Rndm(),  
                    r.Rndm(), r.Rndm() );  
        ttree->Fill();  
    }  
    ttree->Write();  
  
    file.Close();  
    return 0;  
}
```

► Readout:

```
int TTree_TVector_read () {  
  
    TVector3 v3;  
    TVector3* pv3 = &v3;  
  
    TLorentzVector vL;  
    TLorentzVector* pvL = &vL;  
  
    TFile f ("TTree_TVector.root");  
  
    TTree* ttree = (TTree*) f.Get ("ttree");  
    ttree->SetBranchAddress ("v3", &pv3);  
    ttree->SetBranchAddress ("vL", &pvL);  
  
    for (int evt=0; evt < ttree->GetEntries(); evt++)  
    {  
        ttree->GetEvent (evt);  
  
        cout << "[" << evt << "]: [  
                << fixed << setprecision (3) <<  
                << v3[0] <<" : " << v3[1] << " : "  
                << v3[2] << "]" << "\t";  
  
        cout << "[" << vL[0] << " : " << vL[1]  
                << " : " << vL[2] << " : " << vL[3]  
                << "]\n";  
    }  
    f.Close();  
  
    return 0;  
}
```

Notice: Methods of TVector3 and TLorentzVector classes work. E.g.: tree->Draw ("v3.Mag()")

TTree cont. Events with variable number of particles (the simplest way)

► Storage:

```
int TTree_EventManyParticles () {
    Int_t Npart;
    Int_t det[500];
    Float_t energy[500] , time[500];

    TFile f ("manyparticles.root", "RECREATE");
    TTree t ("tree", "My tree");
    t.Branch ("Npart", &Npart, "Npart/I");
    t.Branch ("Det" , det , "det[Npart]/I");
    t.Branch ("Time" , time , "time[Npart]/F");
    t.Branch ("En" , energy, "energy[Npart]/F");

    TRandom3 r; r.SetSeed ();
    for (int ievt=0; ievt < 100 ; ievt++)
    {
        Npart = r.Integer(6);
        cout << "Event " << ievt
             << " has " << Npart << " particles.\n";
        for (int ipart=0; ipart<Npart; ipart++)
        {
            det [ipart] = r.Integer (24);
            time [ipart] = r.Rndm() * 20.;
            energy[ipart] = r.Rndm() * 30.;
            cout << setw(10) << det [ipart]
                 << setw(12) << time [ipart]
                 << setw(12) << energy[ipart] << endl;
        }
        t.Fill ();
    }
    t.Write();
    return 0;
}
```

► Readout:

```
int TTree_EventManyParticles_read () {
    Int_t Npart;
    Int_t det[500];
    Float_t energy[500] , time[500];

    TFile f ("manyparticles.root", "READ");
    TTree* t = (TTree*) f.Get ("tree");
    t->SetBranchAddress ("Npart", &Npart );
    t->SetBranchAddress ("Det" , det );
    t->SetBranchAddress ("Time" , time );
    t->SetBranchAddress ("En" , energy );

    cout << "* This tree has "
         << t->GetEntries() << " entries.\n\n";

    for (int ievt=0; ievt<t->GetEntries(); ievt++)
    {
        t->GetEntry (ievt);
        cout << "* Event " << ievt
             << " has " << Npart << " particles:\n";

        for (int ipart=0; ipart<Npart; ipart++)
        {
            cout << setw(10) << det [ipart]
                 << setw(12) << time [ipart]
                 << setw(12) << energy[ipart] << endl;
        }
    }
    return 0;
}
```

Drawback: it's necessary to predefine the dimension limit (here: 500). Dynamic memory allocation does not work.

TTree cont. Trees with user-defined objects

Event

Branch: Object { Det (int) , Time (float) , Energy (float) }



The implementation recipes changed throughout ROOT versions.

Method suggested for ROOT 5,6 : via **ACLiC** mechanism. Below – demonstrator code for a minimal object.

1. Create a header file `myClass.h`

```
#ifndef __myClass__
#define __myClass__
#include "TObject.h"

class myClass : public TObject {
public:
    Int_t det;          // det
    Double_t ToF;        // ToF
    Double_t Energy;     // Energy

    myClass() { det = 0;
                ToF = 0.; Energy = 0.; }

    // Declarations of our other methods

    ClassDef (myClass,1) // My simple class
};
#endif
```

without semicolon

2. Create the class source code `myClass.cxx`:

```
#include <iostream.h>
#include <myClass.h>

ClassImp(myClass)

// Implementations of our other methods
```

without semicolon

- Class must inherit after `TObject`.
- It must contain the `()` constructor.
- `ClassDef` and `ClassImp` are the preprocessor macros, which paste here the additional builtin methods, e.g. enabling the storage of object in a TTree (`::Streamer`) or creating the documentation.

TTree cont. Trees with user-defined objects

3. Encoding the TTree, which for every event stores 1 object of `myClass` class.

```
#ifdef __CINT__
#else
#include "myClass.h"
#endif

int TTree_myObject ()
{
    if (!TClass::GetDict("myClass"))
        gROOT->ProcessLine (".L myClass.cxx+");

    TRandom3 r; r.SetSeed ();
    myClass* myObj = new myClass ();

    TFile f ("myobjs.root", "recreate");
    TTree* t = new TTree ("tree", "My Tree");
    t->Branch ("myObj", &myObj, 8000, 0);

    for (int evt = 0; evt < 100; evt++) {
        myObj->det      = r.Integer (24) ;
        myObj->ToF      = r.Rndm() * 20. ;
        myObj->Energy   = r.Rndm() * 30. ;
        t->Fill();
    }
    t->Write();
    t->Print();
    f.Close();
    return 0;
}
```

← Enables both ROOT 5/6 versions

4. Encoding the readout of such a TTree.

```
#include "myClass.h"

int TTree_myObject_read ()
{
    myClass* myObj = new myClass ;

    TFile f ("myobjs.root");
    TTree* t = (TTree*) f.Get ("tree");
    t->SetBranchAddress ("myObj", &myObj);

    cout << "This tree has "
          << t->GetEntries() << " events.\n";
    for (int evt=0; evt < t->GetEntries(); evt++)
    {
        t->GetEntry (evt);

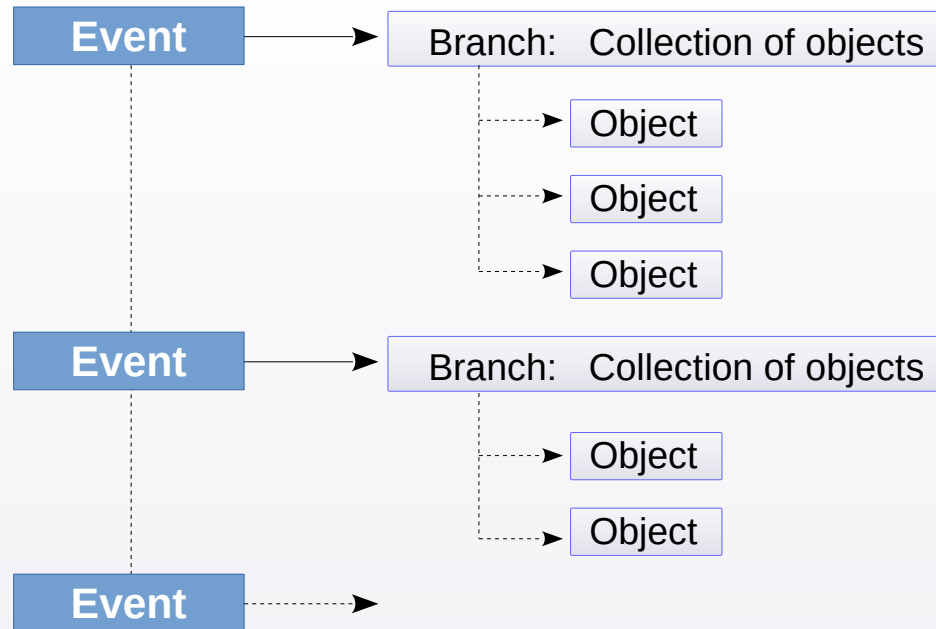
        cout << "[Event " << evt << "]" :  ["
          << setw( 4) << myObj->det
          << setw(12) << myObj->ToF
          << setw(12) << myObj->Energy
          << " ]\n";
    }
    f.Close();

    return 0;
}
```

The `.L` command will create 2 files on the current path:

- `myClass_cxx.so` (compiled object – *shared library*)
- `myClass_cxx.d` (“dependencies”; information for ROOT)

TTree cont. Collection of objects (of the same class) stored in a TTree event



► In order to encode such a structure, we'll use the object of `TClonesArray` class.

It is an array of objects of the same class. These objects must inherit after `TObject`.

A created `TClonesArray` is given the default initial size: 1000 objects.

Once we insert an element at higher position, the enlargement of dimension is done automatically.

► Nb. ROOT features several kinds of arrays for objects (so-called `Collections`).
E.g., within `TOrdCollection` one can store objects of different classes (inheriting after `TObject`).

► On the next slides: exemplary codes that save and read such a "structure".

TTree cont. Collection of objects (of the same class) stored in a TTree event

▶ Exemplary code to store the TCloneArrays in the TTree entries:

- Create the TClonesArray object, giving the name of class of elements. Creating also the pointer to TClassArray.

If we don't set the size, the default size will be 1000 elements.
If we overfull, the array will resize.

- In branch's definition we give the pointer to the pointer of TClassArray.

- Cleaning the array

- Creating a new object of myClass class will automatically store it in the array at a given position.

```
#ifdef __CINT__
#else
#include "myClass.h"
#endif

int TTree_TClonesArray ()
{
    if (!TClass::GetDict("myClass"))
        gROOT->ProcessLine (".L myClass.cxx");

    TFile f ("clonesarray.root", "recreate");

    TClonesArray *myArrayPtr = new TClonesArray ("myClass");
    myClass* myObjectPtr;

    TTree *t = new TTree ("tree", "My Tree");
    t->Branch ("ObjClones", &myArrayPtr, 256000, 0);

    TRandom3 r;  r.SetSeed();

    for (int evt=0; evt<100; evt++) {
        myArrayPtr->Clear();
        int Npart = rand() % 6;

        cout << "Event " << evt << " has " << Npart << " particles. \n";
        for (int iPart = 0; iPart < Npart; iPart++)
        {
            myObjectPtr = (myClass*) myArrayPtr->ConstructedAt (iPart);
            myObjectPtr->det      = r.Integer (24);
            myObjectPtr->ToF      = r.Rndm();
            myObjectPtr->Energy = r.Rndm();
        }
        t->Fill();
    }
    t->Print();  t->Write();
    f.Close();
    return 0;
}
```

TTree cont. Collection of objects (of the same class) stored in a TTree event

▶ Exemplary code to read out the arrays of objects from TTree entries:

- Create 1x TClonesArray through pointer. •→
- We connect to a branch, giving the pointer to the pointer to the TClonesArray object. •→
- Clearing array before event readout •→
- If we get the event, the TClonesArray object is filled automatically. •→
- Iteration over array elements. •→

```
#include "myClass.h"

int TTree_TClonesArray_read ()
{
    TFile f ("clonesarray.root");

    TClonesArray* myArrayPtr = new TClonesArray ("myClass");

    TTree* t = (TTree*) f.Get ("tree");
    t->SetBranchAddress ("ObjClones", &myArrayPtr);

    myClass* myObjectPtr;

    for (int evt = 0 ; evt < t->GetEntries() ; evt++)
    {
        myArrayPtr->Clear();
        t->GetEvent (evt);
        int Npart = myArrayPtr->GetEntries() ;

        cout << "\nEvent " << evt << " has "
              << Npart << " particles: \n";

        for (int iPart = 0; iPart < Npart; iPart++)
        {
            myObjectPtr = (myClass*) myArrayPtr->At (iPart);

            cout << " [" << setw (2) << myObjectPtr->det
                  << ": " << setw (9) << myObjectPtr->ToF
                  << " , " << setw (9) << myObjectPtr->Energy << " ] \n";
        }
    }

    f.Close();
    return 0;
}
```

Finding the root of function

▶ ROOT contains **numerical algorithms**, borrowed from the **GSL library**. In script we'll consider 2 of them.

▶ We start the root search by deciding if it's enough that method uses only the function (e.g. **bisection**), or a derivative is needed (e.g.. **Newton**).

① We write function (+ derivative if needed)

② We put function (+derivative if needed) into a special object called "wrapper".

③ We create the **RootFinder** tool, and set the type of algorithm. Here we use: **bisection** and **Newton**. Full set of methods is available [here](#).

④ We link the tool and the functions

⑤ We evaluate the root finder.

⑥ A result is given by `Root()` method.

```
#include <Math/RootFinderAlgorithms.h>
#include <Math/RootFinder.h>
#include <Math/Functor.h>

using namespace ROOT::Math;

double myfunc (double x) {
    return 3*x - 10;
}

double myfunc_deriv (double x) {
    return 3 ;
}

void macro_RootFinder ()
{
    Functor1D f ( &myfunc );
    RootFinder k ( RootFinder::kGSL_BISECTION ) ;
    k.SetFunction (f, 1, 10);
    k.Solve ();
    cout << "Root via bisection: " << k.Root()
         << endl;

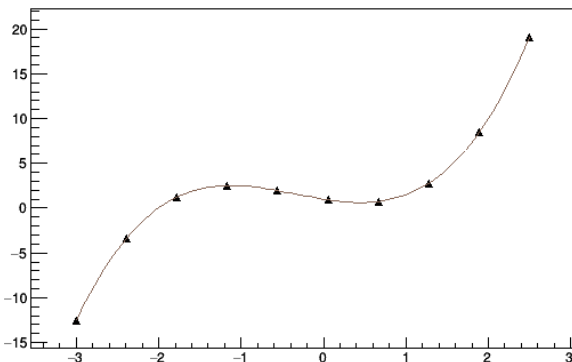
    GradFunctor1D g ( &myfunc , &myfunc_deriv );
    k.SetMethod ( RootFinder::kGSL_NEWTON );
    k.SetFunction ( g , 4. );
    k.Solve ();
    cout << "Root via Newton    : " << k.Root ()
         << endl;
}
```

Interpolation between points

► Available tool:
ROOT::Math::Interpolator,
borrowed from the **GSL** library.

► Algorithm steps:

- ① Store your data points in
double* or vector<double> arrays
- ② Create the Interpolator object,
giving the interpolation type:
 - kLINEAR
 - kPOLYNOMIAL
 - kCSPLINE
 - kCSPLINE_PERIODIC
 - kAKIMA
 - kAKIMA_PERIODIC
- ③ Pass your data using `SetData` method
- ④ Values of interpolation function are
available immediately via `Eval` method



```
void macro_interpolation ()
{
    float xmin = -3, xmax = 2.5;
    Int_t Ndata = 10;
    double xi[Ndata], yi[Ndata];

    TF1* funPoly = new TF1 ("fp",
        "[0]+[1]*x+[2]*x^2+[3]*x^3", xmin, xmax);
    funPoly->SetParameters (1, -1.5, 1, 1);

    for (int i = 0; i < Ndata; i++) {
        xi[i] = i * (xmax - xmin) / (Npts-1) + xmin;
        yi[i] = funPoly->Eval ( xi[i] );
    }

    ROOT::Math::Interpolator inter ( Ndata ,
        ROOT::Math::Interpolation::kPOLYNOMIAL);

    inter.SetData (Npts, xi, yi);

    int Nprobes = 100;
    double Xint[Nprobes], Yint[Nprobes];

    for (int i = 0; i < Nprobes; ++i ) {
        Xint[i] = i*(xmax-xmin)/(Nprob-1) +xmin;
        Yint[i] = inter.Eval ( Xprob[i] );
    }

    TGraph* gf = new TGraph (Npts, xi, yi);
    gf->Draw ("AP");

    TGraph* gi = new TGraph (Nprob, Xprob, Yinter);
    gi->Draw ("SAME L");
}
```

Compilation of standalone C++ code with ROOT functionality



Necessary steps

1. It should be a “decent”, compilable code.
E.g. should contain the `main` function.
2. In the code we have to include all the headers corresponding to used ROOT objects, e.g.:

`#include "TH1F.h"`
3. If we use graphics, we should add the **TRint** graphical interface. In order to do that,
 - include the `TRint.h` header
 - Declare the `main` function with the input arguments:

```
int main (int argc, char* argv[] )
```

- In the `main` function we create the `TRint` object

```
TRint myRint ("myRint", &argc, argv);
```

4. Compilation with “typical” tools – via :

```
g++ code.C `root-config --cflags --libs`
```

In case of extra libraries, we add them at the end, e.g.

```
-lMathMore for Root::Math or
```

```
-lSpectrum for TSpectrum.
```



Exemplary code in C++ : fit of TF1 to TGraph
We compile it as above.



Cstandalone_fitTGraphErrors.C

```
#include "TF1.h"
#include "TGraphErrors.h"
#include "TMath.h"

#include "TRint.h"          // graphics interface
#include "TCanvas.h"

using namespace std;

Double_t myFun (Double_t* xarg, Double_t* par)
{
    Double_t x = xarg[0] , result = 0.;

    for (int st=0; st<=3; st++)
        result += par[st] * TMath::Power (x, st);

    return result;
}

int main (int argc, char* argv[])
{
    TRint myRint ("myRint", &argc, argv);
    TCanvas* can1 = new TCanvas ("can1",
                                "can1", 600, 400);

    TGraphErrors gr ("dataPoints.txt");
    gr.SetTitle ();

    TF1 fun ("fun", myFun , -3, 5, 4);
    fun.SetParameters (-1. , 1. , -6. , 1.);
    gr.Fit ( &fun , "" );
    gr.Draw ("AP");

    can1->Update();
    cin.ignore();
    return 0;
}
```

Compilation through *make*

Within Linux, many applications are installed from sources using *make*.

The aim of *make* is the compilation and, if needed, linking of the package.

You can see the minimal *make* macro for the code from previous page. It doesn't perform linking, but has options for linking to ROOT libraries ready.

makefile

```
CC=g++
CFLAGS=`root-config --cflags --libs`
LDFLAGS=`root-config --glibs`

SOURCE=Cstandalone_fitTGraphErrors.C
TARGET=Cviamake_fitTGraphErrors

Cviamake_fitTGraphErrors: $(SOURCE)
    $(CC) -o $(TARGET) $(SOURCE) $(CFLAGS)

clean:
    rm -f ./.*~ ./.*.o ./Cviamake_fitTGraphErrors
```

Unification of code

Solutions are available for a common code, which handles two variants of launching:

- ① as a compilable code
(eg. via g++ with ROOT flags)
- ② as a macro in interactive session.

One of solutions is the usage of `#if defined` preprocessor commands.

The demonstrator code shows also,
– how to handle the input arguments
– where to place `#include` headers.

```
#if defined __CINT__ || defined __CLING__

int macro_cprogram_unifier (int InputValue = 123) {
    cout << "\n Hello, I am being interpreted." << endl;

#else

#include "TMath.h"
#include <iostream>
#include <iomanip>
using namespace std;

int main (int argc, char* argv[]) {
    cout << "\n Hello, I was compiled." << endl;
    int InputValue = (argc > 1) ? atoi (argv[1]) : 123;

#endif

    cout << "\n Okay, and this is the common portion of code.";
    cout << "\n TMath::Pi() = " << setprecision (18) << TMath::Pi();
    cout << "\n Input value (default: 123) = " << InputValue;
    cout << "\n\n";
    return 0;
}
```