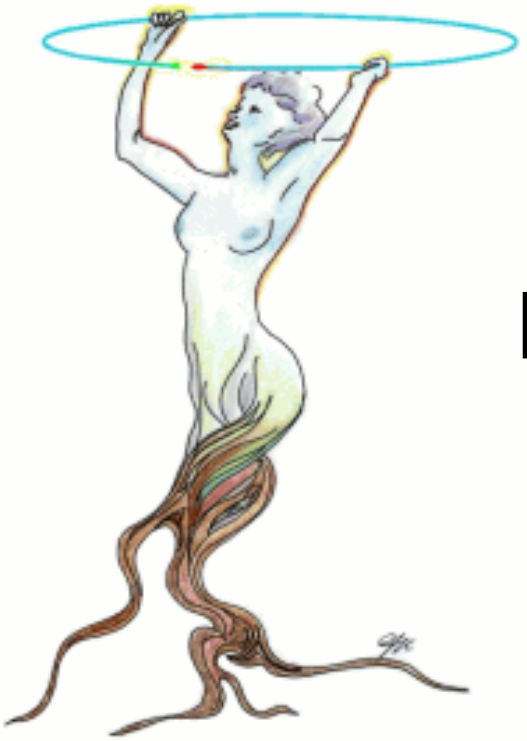
ROOT

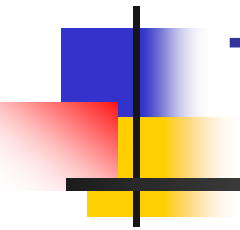
An Object-Oriented
Data Analysis Framework



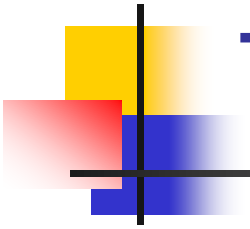
Part 2

Luciano Pandola
INFN, LNS

Thanks to: N. Di Marco, S. Panacek and A. Tramontana



TGraphs and TProfiles



TGraph & Co.

- An other basic object of ROOT: **2D scatter plots**
- **`TGraph`** stores a set of points in (x,y)
  - **`TGraphErrors`**: error bars
  - **`TGraphAsymmErrors`**: asymmetric error bars
  - **`TGraphBentErrors`**: asymmetric error bars in diagonal directions
  - **`TGraphPolar`**: polar scale ...
- Many commands in common with the histograms and functions
  - Fit(), Draw()

Filling a TGraph* - 1

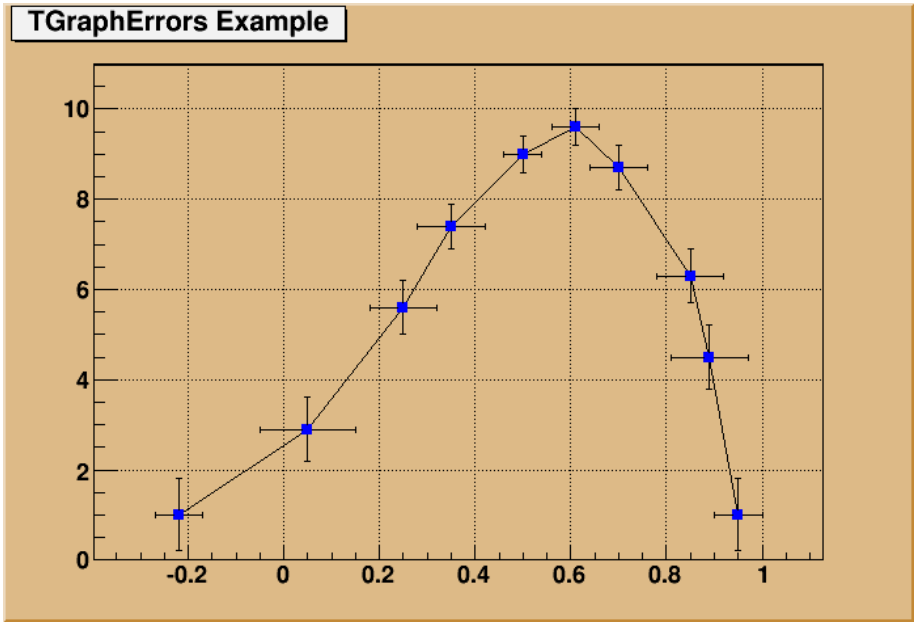
- Option #1: give each point individually

```
Int_t n = 10;
TGraphErrors* gr = new TGraphErrors(n);
for (Int_t i=0;i<n;i++)
{
  gr->SetPoint(i,x,y);
  gr->SetPointError(i,x,y)
}
gr->Draw("AZP");
```

- Points can be edited/moved also in the GUI

Filling a TGraph* - 2

- Option #2: feed the vectors of points and errors



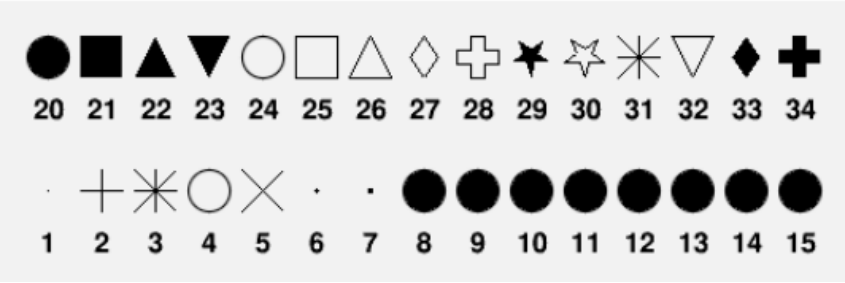

```
 Int_t n = 10;
 Double_t x[n]  = {-0.22, 0.05, 0.25, 0.35, 0.5,
0.61,0.7,0.85,0.89,0.95};
 Double_t y[n]  = {1,2.9,5.6,7.4,9,9.6,8.7,6.3,4.5,1};
 Double_t ex[n] = {.05,.1,.07,.07,.04,.05,.06,.07,.08,.05};
 Double_t ey[n] = {.8,.7,.6,.5,.4,.4,.5,.6,.7,.8};
 gr = new TGraphErrors(n,x,y,ex,ey);
```

TGraph options

- Can set dimension, type and color of the marker

```
gr->SetMarkerColor(4);
gr->SetMarkerStyle(21);
```

- Drawing options given from **`TGraphPainter`**
  - Can do exclusion plots or similar
  - Check documentation for all options
  - Can be done by the GUI

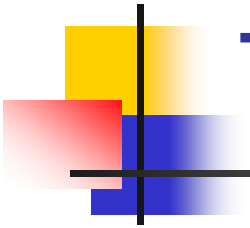


Drawing TGraphs

- Use the same commands as histograms to:
  - draw, set title, set axis labels and options (e.g. time scale), fit, ...

```
gr->SetTitle("My Graph");
gr->GetXaxis()->SetTitle("energy (keV)");
gr->Fit("gaus");
gr->Draw("APL")
```

Notice: axes are *not drawn by default*. The **option A** must be specified to have them. If not given, the graph is drawn in the current coordinate system

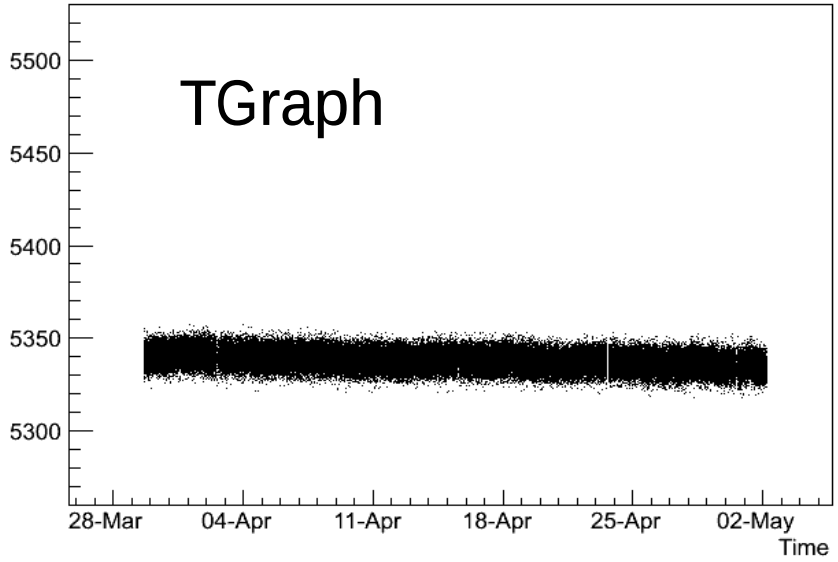


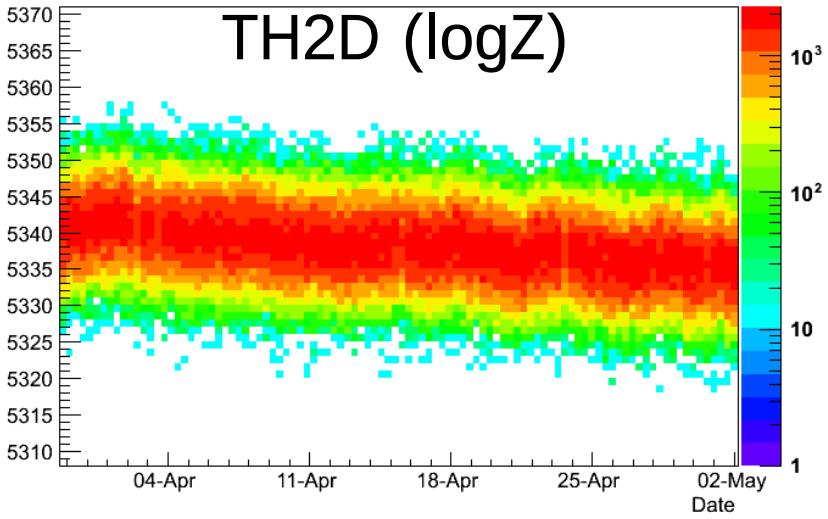
TProfile

- Profile histograms (TProfile) alternative to 2D-histograms
- Display the mean value of y and its error for each bin in x
  - Error: standard error on the mean (rms/$\sqrt{N}$)
  - It is possible also to show the global rms as error
- Useful when you want to see the general trend of y vs. x
  - It makes sense when y is an unknown (but **single-valued**) approximate function of x (apart from statistical fluctuations)

TProfile – when to use it

- Real-life case:
  - Monitor the stability of a DAQ system, a constant-amplitude test pulse is injected every 10 or 20 s. The measurement lasts for weeks
- How do we plot amplitude vs. time and identify variations?



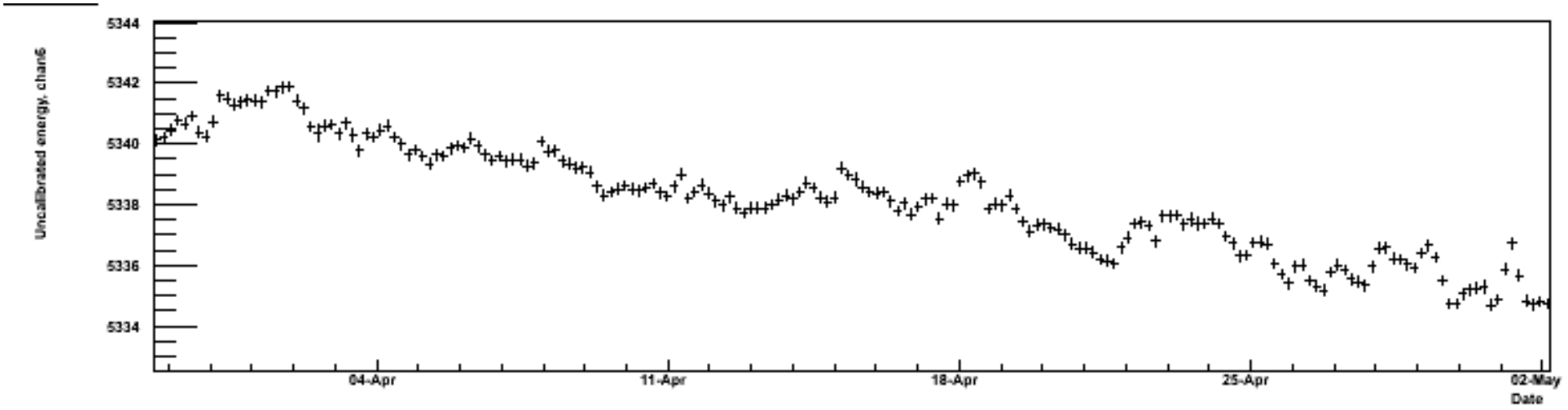


TProfile – when to use it

- Generate a **TProfile** from the TH2D

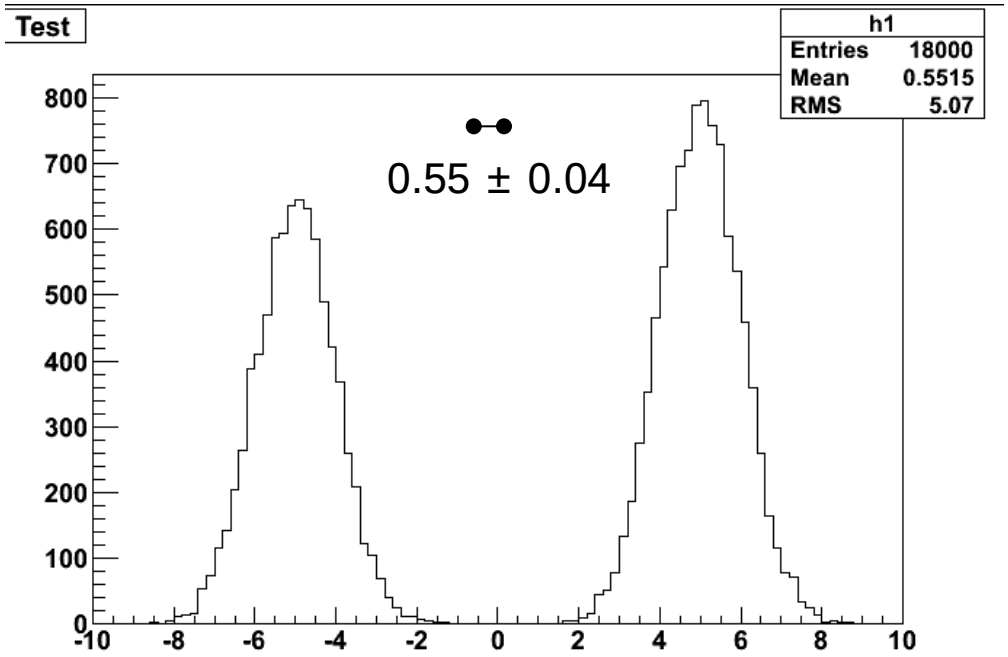
```
TH2D* h2 = …;
TProfile* prof = h2->ProfileX();
prof->Draw()
```

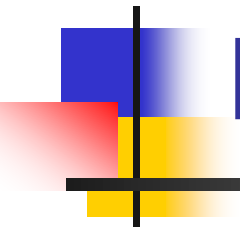
- Can use the same tools as for histograms (e.g. **Fit**)



TProfile – a caveat

- If the distribution is not "single-valued", you can get anomalous values or error bars
  - E.g. if the projection y(x) has two peaks

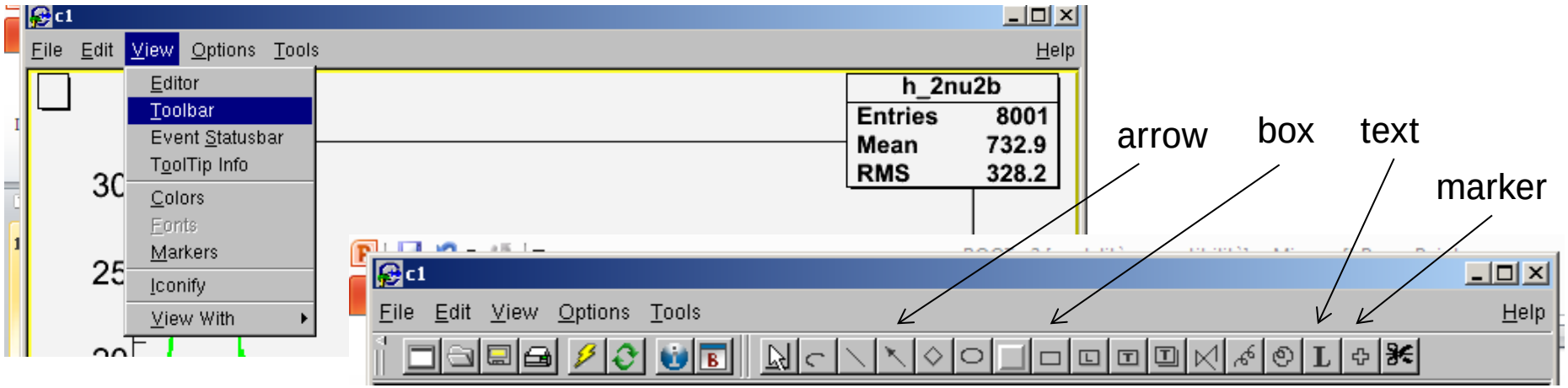




Extra useful tools

TText and other utilities

- Can write text in any ROOT canvas
  - Can also draw lines, arrows, boxes, arcs...
  - **`TLine, TArrow, TBox, TArc`**
- Easy to do interactively with the GUI, from the **toolbar**
- Of course one can select font, style, size and color



TText and other utilities

- Everything is also doable via command line or macro

```
TText* text = new TText(10,20,"Ciao");
text->SetTextFont(42);
text->SetTextSizePixels(16);
text->SetTextColor(kRed);
text->Draw("same");
```

x y

- Graphical options managed by **TAttText**
  - Alignment, angle, color, size, font
- Same for other objects

```
TArrow* ar2 = new TArrow
(0.2,0.1,0.2,0.7,0.05,"|>");
ar2->SetAngle(40);
ar2->Draw();
```

x1, y1, x2, y2, size, style

TLatex

- Latex-style math and tools are also supported
  → \ replaced by #

```
h->GetYaxis()->SetTitle("#chi^{2}");
```

  χ^2

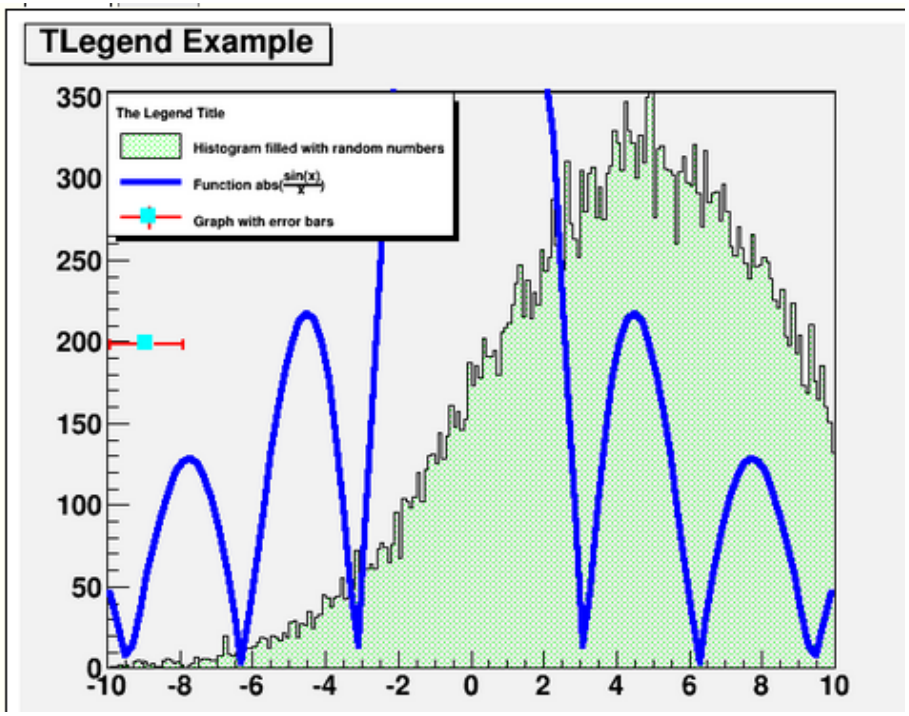
- Can be also used in a similar way as TText

```
TLatex lat;
lat.SetTextAlign(12);
lat.DrawLatex(x, y, "#frac{#pi}{2}");
```

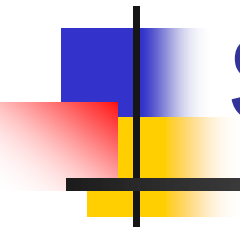
- Usual GUI functionalities

TLegend

- It produces a legend (that can be drawn on the current canvas)
  - Applies to all kinds of ROOT objects (functions, graphs, histograms)
- Can decide which attribute (marker/line/fill) is displayed for each
  - Follows automatically the changes of attributes




```
TLegend leg(0.1,0.7,0.48,0.9);
leg.SetHeader("Title");
leg.AddEntry(h1,""Histogram,"f");
leg.AddEntry("f1","Function","l");
leg.AddEntry("gr","Graph ","lep");
leg.Draw();
```



Scripts and C++

Convention on coding and names

Based on Taligent rules

Classes	Start with T	TTree, TBrowser
Non-class types	End with _t	Int_t
Class data members	Start with f	fTree
Class methods	Start with capital letter	Loop()
Constants	Start with k	kInitialSize, kRed
Static variables	Start with g	gEnv
Class static data members	Start with fg	fgTokenClient

The scripts – unnamed scripts

- Suitable for **very small tasks**
  - Start with "{" and end with "}"
  - All variables in global scope
  - No definition of classes and functions
  - No input parameters

Unnamed script: hello.C

```
{
  cout << "Hello" << endl;
}
```

The scripts – named scripts

- Suitable for more complex tasks, which still do not require an ad-hoc executable (it is a macro!)
  - C++ functions
  - Scope rules according to the standard C++
  - The function has the **same name as the file**. It can be executed (**interpreted**) with **.x**

```
root [3] .x myMacro.C
```

  - Supports input parameters and classes

Named script: say.C

```
void say(TString what="Hello")
{
  cout << what << endl;
}
```

```
root [3] .x say.C
Hello
root [4] .x say.C("Hi")
Hi
```

ACLiC: Automatic Compiler of Libraries for CINT

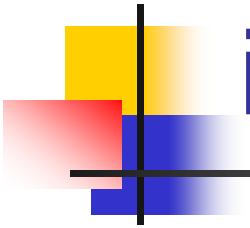
- Named scripts can be **interpreted** (line-by-line) by the CINT interpreter

```
root [3] .x myMacro.C;
```

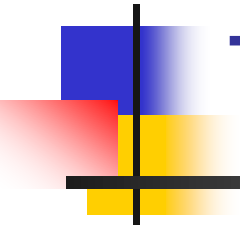
- **Compiled** to produce a shared library via ACLiC (and then possibly executed)

```
root [3] .L myMacro.C++;  //always recompile
root [3] .L myMacro.C+;   //recompile if necessary
root [3] .x myMacro.C++;  //compile and execute
root [3] .L myMacro_C.so; //load the shared library
root [3] myMacro();       //execute the function
root [3] .U myMacro_C.so; //unload the library
```

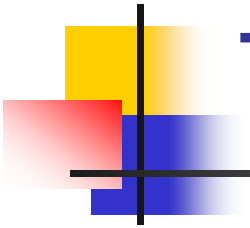
Named scripts: compiled vs. interpreted



- A compiled named script is pratically equivalent to a C++ **executable**
  - Full C++/coding flexibility
  - The syntax is checked by the compiler **prior** to the execution
  - **Much faster** (x5) than the interpreted macro
  - Suitable for very complex tasks
  - The only major difference is that you need to launch it in a ROOT session
- Notice: if you compile a named script, you will need to specify all relevant #include
  - Not required if the script is interpreted
- Suggestion: always compile



The TFile's



TFile

- **TFile** is the ROOT object to handle I/O (**binary**) files
- Optimized to store ROOT objects, support data compression
  - ROOT format used for the raw data (or interchange format) by several experiments
- Can be organized in sub-directories
- Options are:

`NEW or CREATE`	create a new file for writing, if the file already exists the file is **not opened.**
`UPDATE`	open an existing file for writing. If no file exists, it is created
`RECREATE`	create a new file, if the file already exists it will be overwritten
`READ (default)`	open an existing file for reading

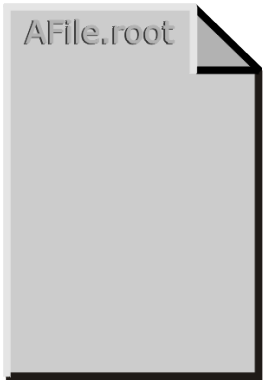
Open a file and get content

- Open a file for reading:

```
root[] TFile f("Example.root")
```

- List the content of the file

```
root[] f.ls()
TFile**         Example.root    ROOT file
 TFile*         Example.root    ROOT file
  KEY: TTree    myTree;1        Example ROOT tree
  KEY: TH1F     totalHistogram;1        Total Distribution
  KEY: TH1F     mainHistogram;1 Main Contributor
  KEY: TH1F     s1Histogram;1   First Signal
  KEY: TH1F     s2Histogram;1   Second Signal
```



- Load/retrieve stored objects by **name**

```
root[] totalHistogram->Draw();
root[] TH1F* myHisto = (TH1F*)
  f.Get("totalHistogram");
```

Works from **command line**, not from macros

General method (need a cast)

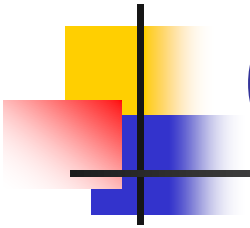
More about TFiles

- When a ROOT file is opened it becomes the current directory
  - Manual switch: `file.cd();`
  - If there are no open files, the current directory is the memory (gROOT)
- Histograms and trees that are created after the file opening are **saved automatically** on it
  - When the file is closed, all ROOT objects associated to it are cleared from the memory
- Any ROOT object which derives from `TObject` (e.g. Graphs, Canvas, Arrows, named parameters) can be written on a ROOT file
  - Must be added explicitly

```
myObject->Write("name");
```

How to use ROOT in other C++ projects (Makefile or cmake)

How to use ROOT in other (external) programs - 1



- ROOT TTree/histograms can be typically generated by DAQ, data analysis or simulations (e.g. a Geant4 application)
- In the real-life it is often necessary to use the ROOT libraries within other **external C++ programs**
  - Needed also if you want to convert a macro into a **stand-alone executable**
- To make the thing work: the Makefile or the compiler command line must contain:
  - Compilation: the path to the ROOT header files (.h)
  - Linking: the path to the ROOT compiled libraries (.so) and the names of the libraries

How to use ROOT in other (external) programs - 2

- A ROOT command (in **$ROOTSYS/bin**) is available which gives back the "**compiler-ready**" options for headers and libraries
  - **root-config –-cflags**

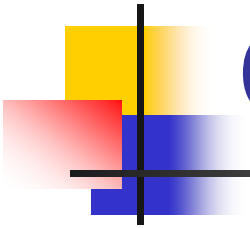
    On my own system, it gives :

    ```
    -pthread -m64 -I/usr/local/root/include
    ```
  - **root-config –-libs**

    On my own system, it gives:

    ```
    -L/usr/local/root/lib -lCore -lCint -lRIO -lNet -lHist
      -lGraf -lGraf3d -lGpad -lTree -lRint -lPostscript -
      lMatrix -lPhysics -lMathCore -lThread -pthread -lm -
      ldl –rdynamic
    ```
- **g++ hello.cc –o hello `root-config --cflags --glibs`**

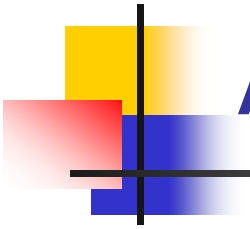
An example: the Geant4 GNUmakefile



- To use ROOT in a Geant4 application, you just add to the Geant4 **GNUmakefile**

```
CPPFLAGS += `root-config --cflags`
LDFLAGS  += `root-config --libs`
```

- **CPPFLAGS** are the compiler **options** for the **compilation** phase, while **LDFLAGS** are the compiler **options** for the **linking** phase
  - In other Makefiles/systems, the names of the flags can be different
- The Geant4 GNUmakefile are deprecated now (will be removed) but the concept is still valid for other applications/Makefiles



An other example: cmake - 1

- The most recent ROOT releases have a ready-for-the-use .cmake configuration file

  $ROOTSYS/cmake/modules

- Also Geant4 has a cmake configuration file for ROOT

  [geant4-build]/Modules/FindROOT.cmake

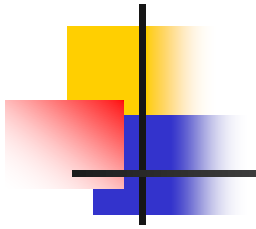
- The directory of the ROOT cmake configuration must be given to the executable cmake via the

  -DCMAKE_MODULE_PATH option

An other example: cmake - 2

- Edit the CMakeLists.txt file
  - Retrieve ROOT, use headers and libraries

```
find_package(ROOT)
if (ROOT_FOUND)
 message("ROOT package found. --> ok ${ROOT_INCLUDE_DIR}")
else()
 message (FATAL_ERROR "ROOT NOT found")
endif()
…
include_directories(${ROOT_INCLUDE_DIR} ${Geant4_INCLUDE_DIR}
${PROJECT_SOURCE_DIR}/include)
…
target_link_libraries(myApplication ${Geant4_LIBRARIES}
${ROOT_LIBRARIES})
```



It is your turn, now:

- Try Task1 under

http://geant4.lngs.infn.it/ROOTCatania2014/introduction/index.html