ROOT : The Package and the Language

Shamona Fawad Qazi
National Centre for Physics

Outline

Practical introduction to the ROOT framework

- Starting ROOT – ROOT prompt
- Macros – Functions
- Histograms – Files
- Trees – TBrowser
- Basics of debugging

Nomenclature

Blue : you type it

Red : you get it

Introduction

What is it?

Very versatile software package for performing analysis on HEP data

- develop and apply cuts on data
- perform calculations & fits
- make plots
- save results in ROOT files

ROOT can be used in many ways:

Command line – good for quickly making plots, checking file contents, etc.

Unnamed macros – execute commands as if you typed them on the command line

list of commands enclosed by one set of { }.

execute from ROOT command line: ".x file.C"

Named macros – best for analysis, can be compiled and run outside of ROOT, or loaded and executed during interactive session

Interactive ROOT uses a C++ interpreter (CINT) which allows (but does not require) you to write *pseudo-C++*

Be careful! This will make your programming much more difficult later in life!

It's best if you try to use standard C++ syntax, instead of the CINT shortcuts.

ROOT CINT syntax allows the following sloppy things:

"." and "–>" are interchangeable

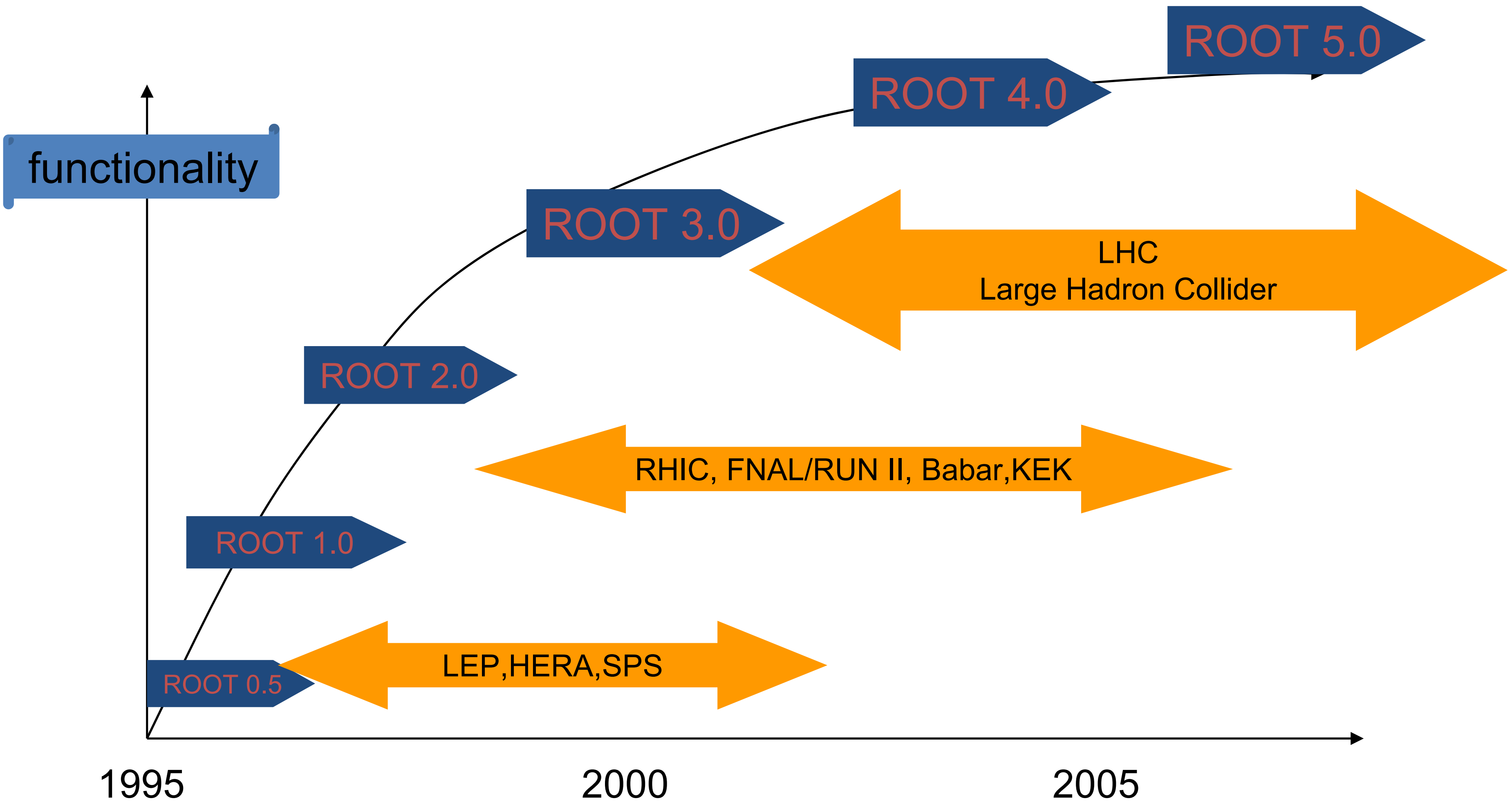
";" is optional at the end of single commands

Many commands may be accessed interactively (point and right-click in plots)

History

- Project Started in 1995
- First release Nov 1995
- 8 full time developers at CERN, plus Fermilab, Agilent Tech, Japan, MIT (one each)
- Large number of part-time developers : let users participate
- Available (incl. source) under GNU LGPL data format
- Used by all HEP experiments

Root Project Evolution



Root in a Nutshell

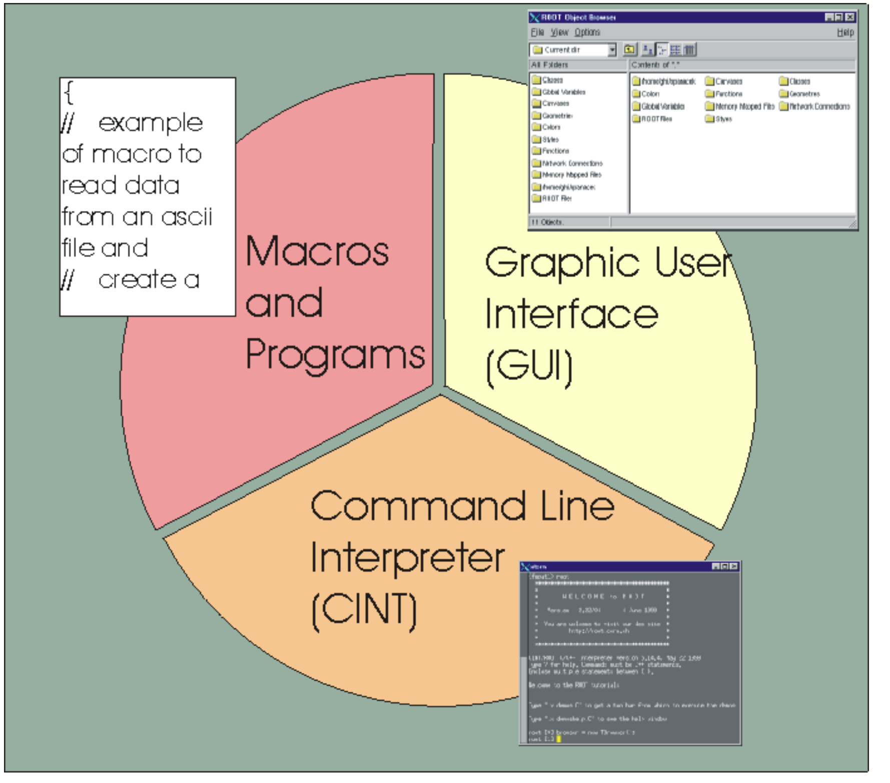
Framework for large scale data handling

Provides, among others

- an efficient data storage, access and query system (Petabytes)
- advanced statistical analysis algorithms (multi dimensional histogramming, fitting, minimization and cluster finding)
- scientific visualization : 2D and 3D graphics, postscript
- PDF, Latex
- Fully cross platform, Unix/Linux, MacOS X and Windows

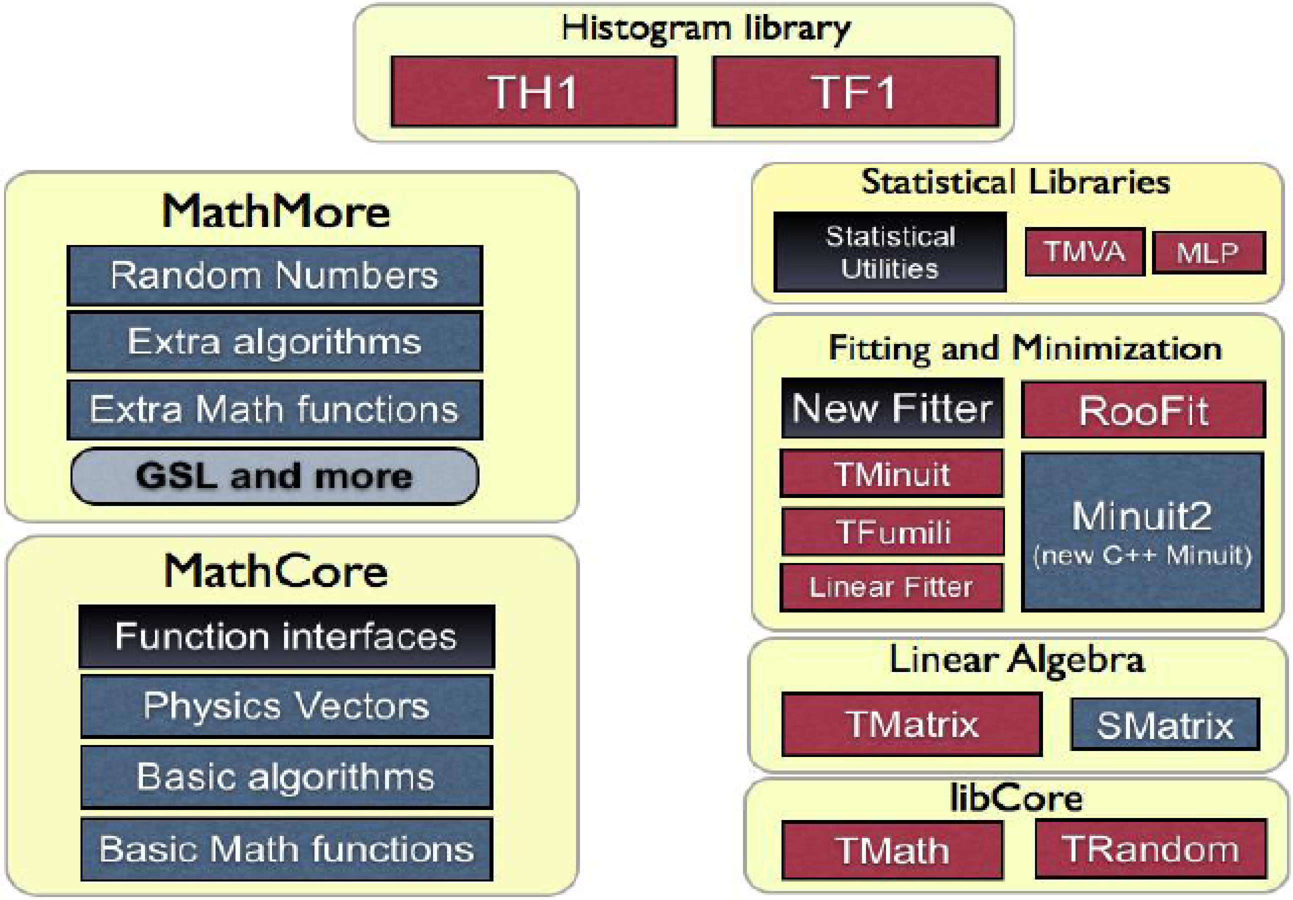
Three User Interfaces

- GUI
  Windows, buttons, menus
- Command line
- Macros, applications, libraries (C++ compiler and interpreter)



TMath

- A very concise math library



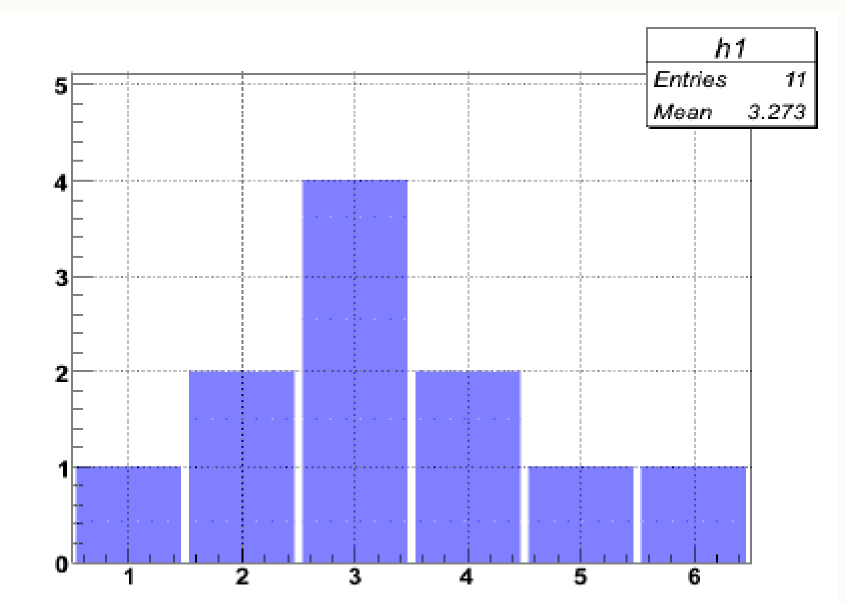
Few Definitions

- Histograms
- Fitting
- Graphics

What is a Histogram?

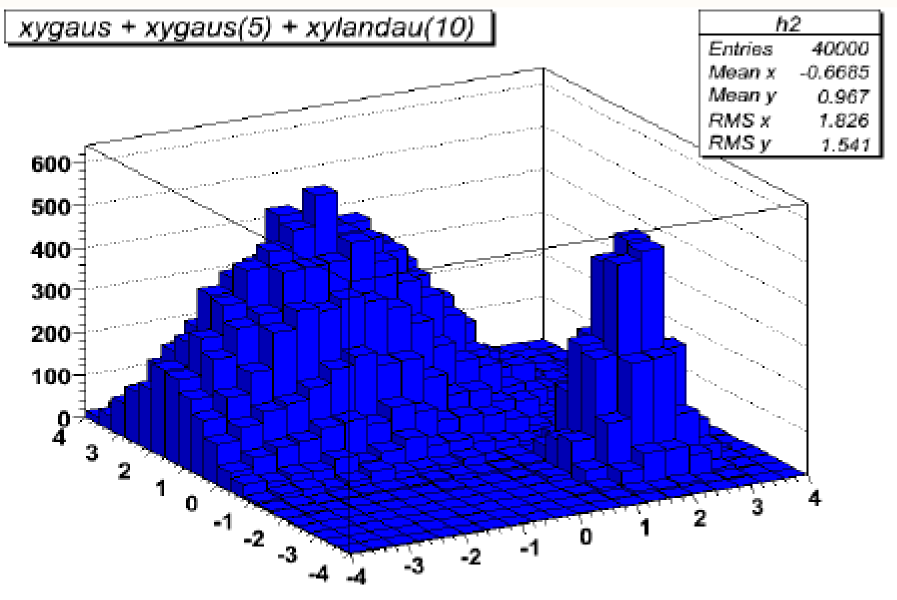
- Histogram is just occurrence counting, i.e. How often same number appears in data

Example: {1,3,2,6,2,3,4,3,4,3,5}

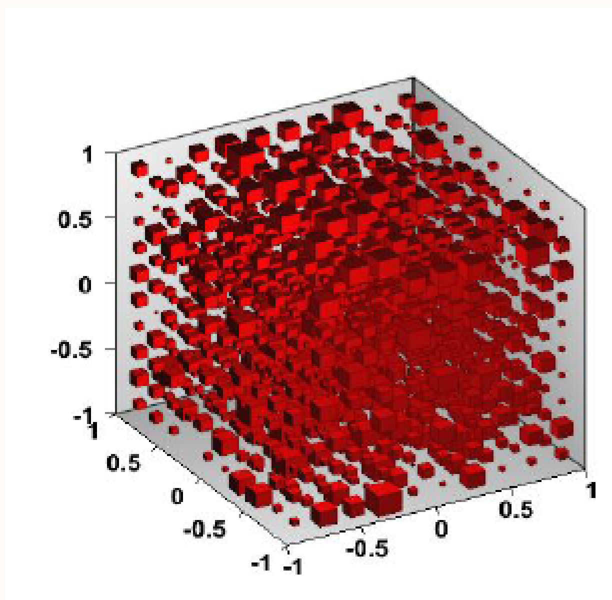


2D / 3D Histograms

- Height represents frequency (2D)
- Volume represents frequency (3D)



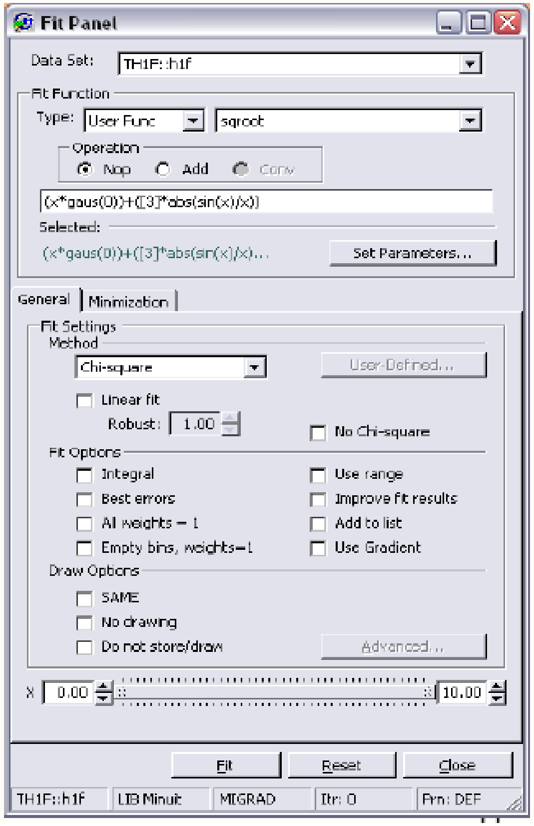
2D Histogram

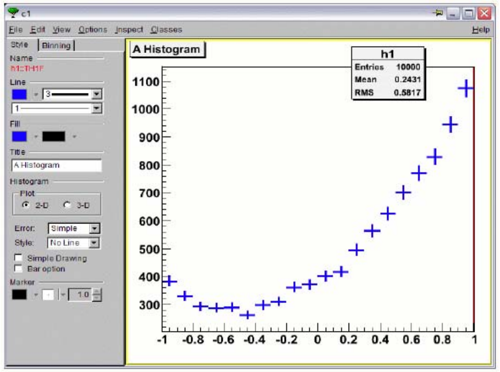


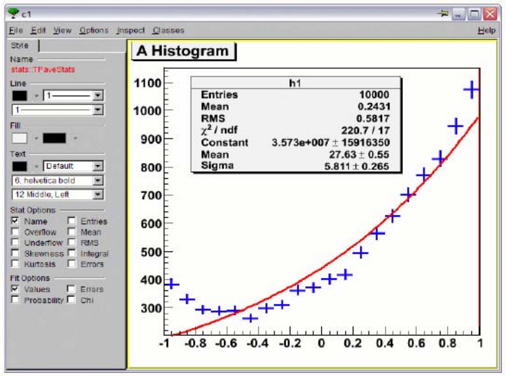
3D Histogram

Fitting

- Finding a function which passes through most of data points
- Can be performed using fit Panel or by writing a macro

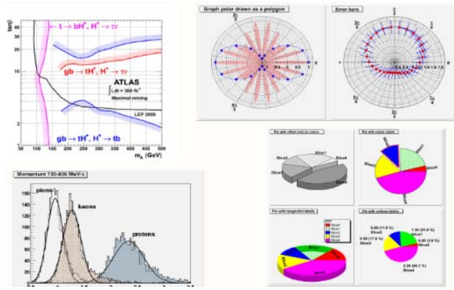




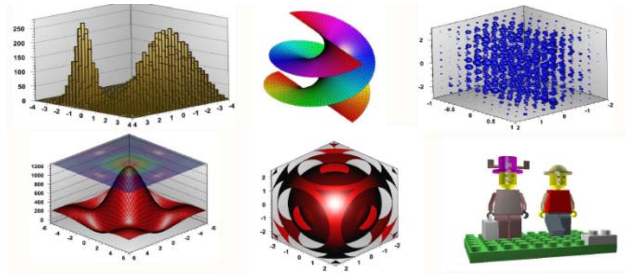


Examples of Visualization

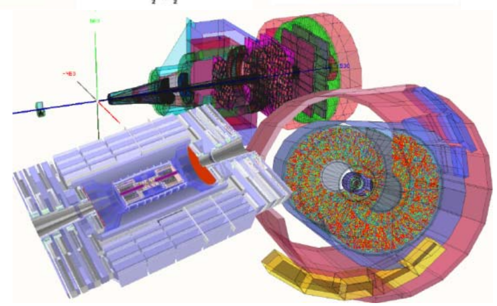
- Some graphs



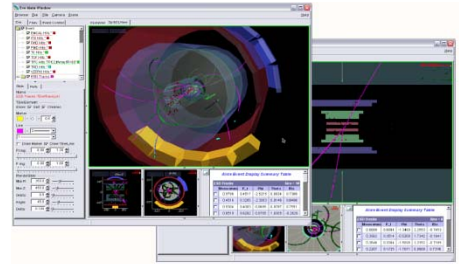
- OpenGL



- Detector Geometry



- EVE: Event Visualization Environment



Setting up ROOT

Before starting ROOT:

- setup environment variables

    $PATH $LD_LIBRARY_PATH

    (ba)sh : $. */PathToRoot/bin/thisroot.sh*

    (t)csh : $ source */PathToRoot/bin/thisroot.csh*

Starting ROOT

    $ root $ root -l (without splash screen)

    root [0]

Command history

– Scan through with **arrow keys**

– Search with **CTRL-R** (like in bash)

Using ROOT

- Root as Pocket Calculator

```
root [0] sqrt(42);
(const double)6.48074069840786038e+00
root [1] double val = 0.17;
root [2] sin(val);
(const double)1.69182349066996029e-01
```

- To run function mycode() in file mycode.C:

```
root [0] .x mycode.C
```

- Equivalent: load file and run function:

```
root [0] .L mycode.C
root [1] mycode()
```

- Quit:

```
root [0] .q
```

Using ROOT-II

- Typing multi-line commands

```
root [ ] for (i=0; i<3; i++)printf(“%d\n”,i)
```

or

```
root [ ] for (i=0; i<3; i++){
end with '}', '@':abort > printf(“%d\n”,i)
end with '}', '@':abort > }
```

- Aborting wrong input

```
root [ ] for (i=0; i<3; i++){
end with '}', '@':abort > @
```

Don’t panic!
Don’t press CTRL-C!
Just type @

Data Types in Root

Signed	Unsigned	Size of [bytes]
Char_t	UChar_t	1
Short_t	UShort_t	2
Int_t	UInt_t	4
float_t	UInt_t	4
Long_t	ULong_t	8
Double_t	UDouble_t	8

Macros

- Combine lines of codes in macro

  Unnamed Macros
    - No parameters
    - For example macro1.C

```
{
   for (Int_t i=0; i<3; i++) printf("%d\n", i);
}
```

- Executing macros

```
root [ ] .x macro1.C
$ root –l macro1.C
$ root –l –b macro1.C (batch mode → no graphics)
$ root –l –q macro1.C (quit after execution)
```

Macros-II

- Named Macros
  - May have parameters
  - For example macro2.C

```
void macro2(Int_t max = 10)
{
   for (Int_t i=0; i<max; i++) printf("%d\n", i);
}
```

- Running named macros

```
root [ ] .x macro2.C
```

- Loading macros

```
root [ ] .L macro2.C
```

- Prompt vs. Macros Loading macros
  - Use the prompt to test single lines while developing your code
  - Put code that is to be reused in macros

Don't forget to change the function name after renaming a macro

Plots for Papers
It is very useful to have all the code that creates a plot in one macro. Do not create "final" plots using the prompt or the mouse (you'll be doing it again and again).

Functions

- The class TF1 allows to draw functions

  root [ ] f = new TF1("func", "sin(x)", 0, 10);

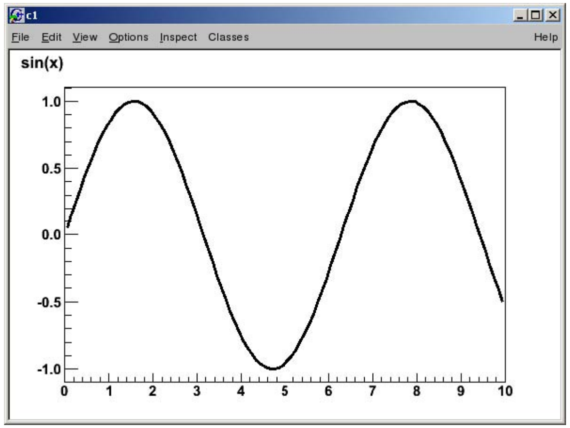
  – "func" is a (unique) name

  – "sin(x)" is the formula

  – 0, 10 is the x-range for the function

  root [ ] f->Draw();

Canvas



- The style of the function can be changed on the command line or with the context menu (→ right click)

  root [ ] f->SetLineColor(kRed);

- The class TF2(3) is for 2(3)-dimensional functions

Histograms

- Contain binned data – probably the most important class in ROOT for the physicist
- Create a TH1F (= one dimensional, float precision)

  root [ ] h = new TH1F("hist", "my hist; Bins; Entries", 10, 0, 10);

  – "hist" is a (unique) name

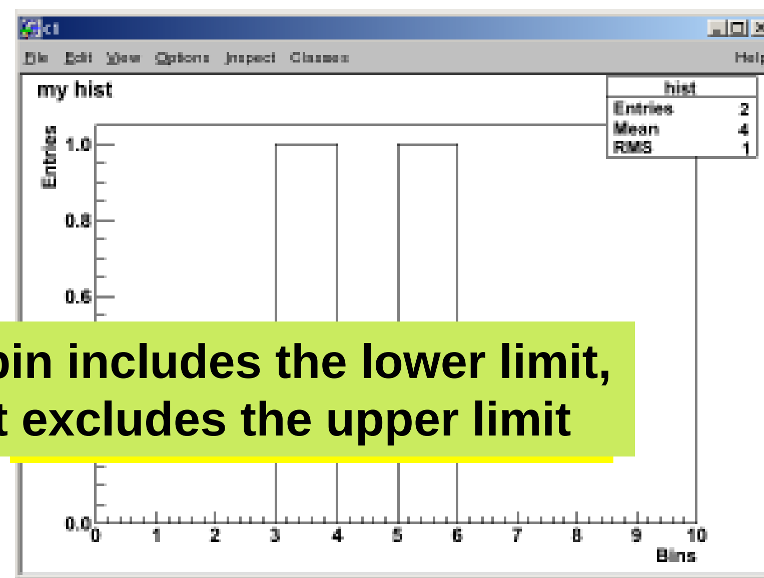
  – "my hist; Bins; Entries" are the title and the x and y-axes labels

  – 10 is the no. of bins

  – 0, 10 are the limits on the x-axis

  Thus the first bin is from 0 to 1,

  the second from 1 to 2 etc.



A bin includes the lower limit, but excludes the upper limit

- Fill the histogram

  root [ ] h->Fill(3.5);

  root [ ] h->Fill(5.5);

- Draw the histogram

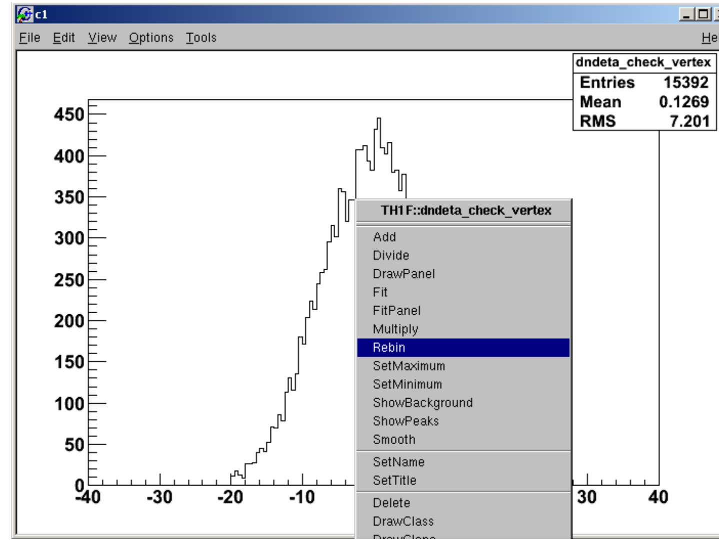
  root [ ] h->Draw();

Histograms-II

- Rebinning

  root [ ] h->Rebin(2);

  Two bins of h will be merged into one

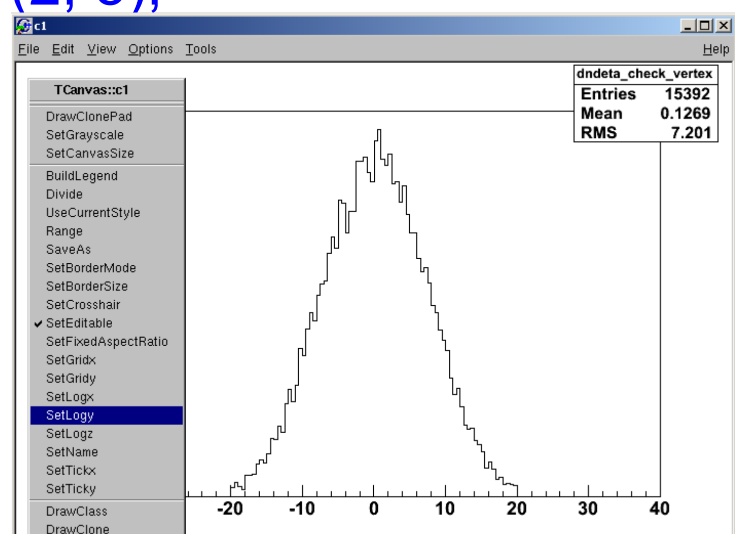


- Change ranges
  - with the mouse
  - with the context menu
  - command line

  root [ ] h->GetXaxis()-> SetRangeUser(2, 5);

- Log-view
  - right-click in the white area at the side of the canvas and select SetLogx (SetLogy)
  - command line

  root [ ] gPad->SetLogy();



Fitting Histograms

- Interactive

  root [ ] f = new TF1("func", "sin(x)", 0, 10);

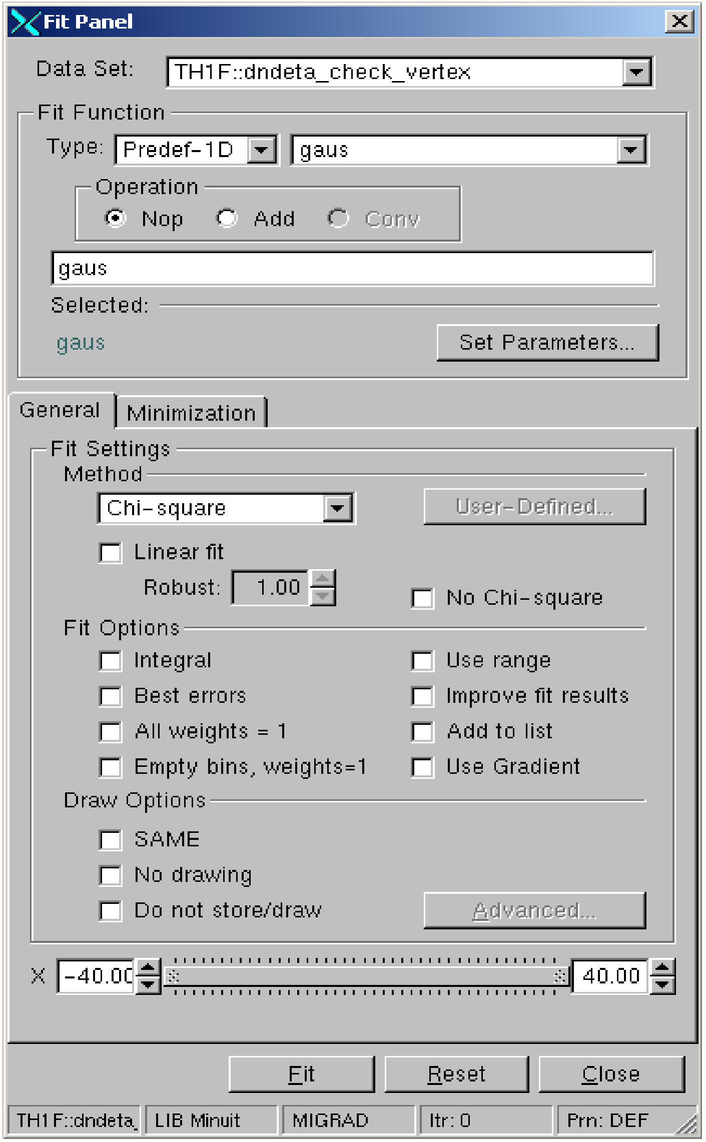
  – Right click on the histogram and choose "fit panel"
  – Select function and click fit
  – Fit parameters
    - ★ are printed in command line
    - ★ in the canvas: options -fit parameters

  root [ ] f->Draw();

- Command line

  root [ ] h->Fit("gaus");

  – other predefined functions, polN (N=0…9), expo etc.



Fitting Histograms-II

- User defined functions
  - Define the function e.g. a single gaussian

$$\frac{1}{\sqrt{2\pi\sigma}} e^{-(x-\mu)^2/2\sigma^2}$$

```
root [ ] f = new TF1("func", "[0] / (2.506628381*[2]) *
exp( ((x - [1])*([1] - x)) / (2*[2]*[2])" );
```

  - Define the parameter names

```
root [ ] f->SetParName( 0, "Constant" );
root [ ] f->SetParName( 1, "Mass" );
root [ ] f->SetParName( 2, "Sigma" );
```

  - Set or fix the values for the parameters

```
root [ ] f->SetParameter( 0, 1600 );
root [ ] f->SetParameter( 1, 0.0 );
root [ ] f->FixParameter( 2, 0.01 );
```

Fitting Histograms-III

- Fit the histogram

```
root [ ] h->Fit("f" );
```

  – Draw the error bars

```
root [ ] h->Draw("e" );
```

  – Define the color of the fit function

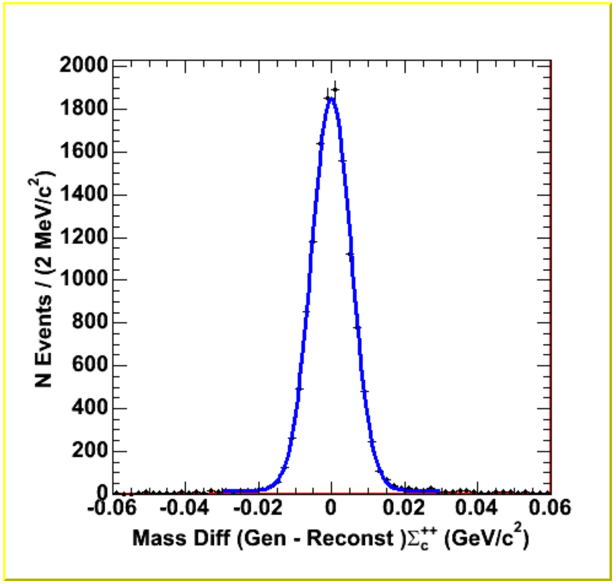
```
root [ ] f->SetLineColor( 2 );
```

- Drawing two histograms on one canvas
  – First fill both histograms h1 and h2

```
root [ ] h1->Draw();
root [ ] h2->Draw(" same" );
```

  – Similar for functions

```
root [ ] func2->Draw(" same" );
```



Gaussian Fit parameters:

$$\text{Area} = 12478 \pm 116$$

$$\text{Mean} = -0.00011 \pm 0.00005 \ \text{GeV/c}^2$$

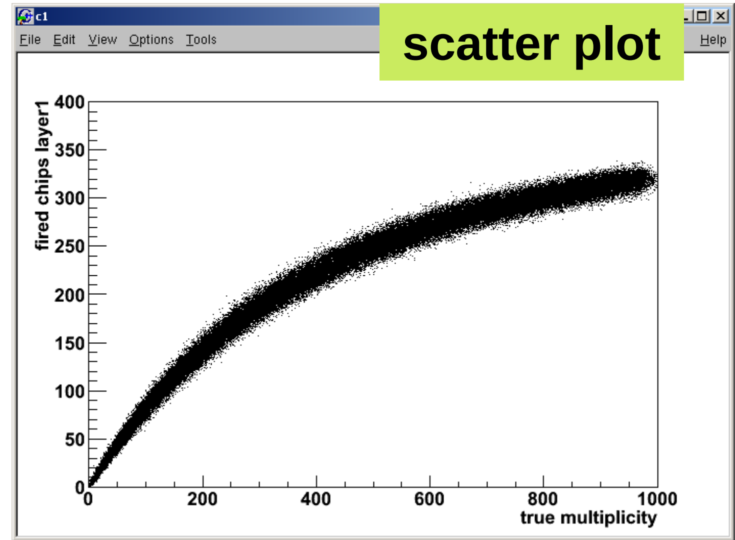
$$\sigma = 0.00543 \pm 0.00004 \ \text{GeV}$$

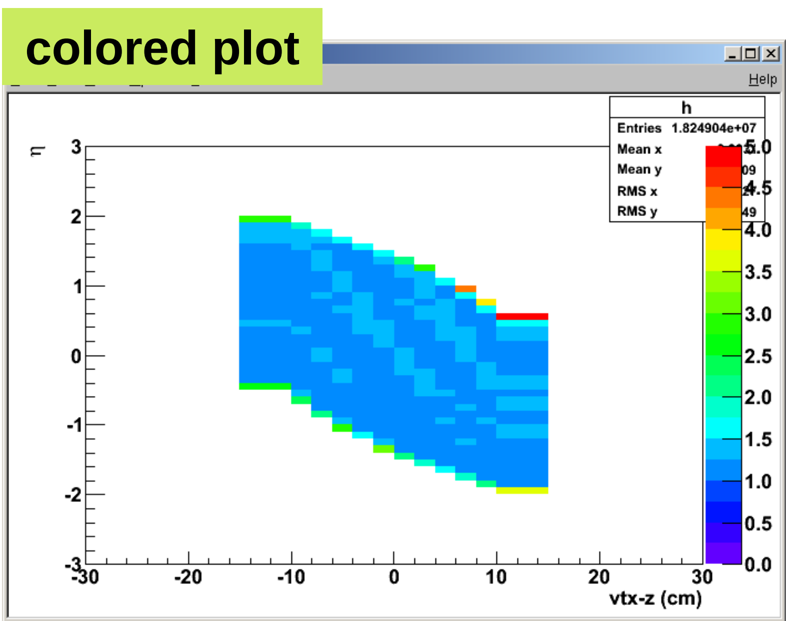
2D Histograms

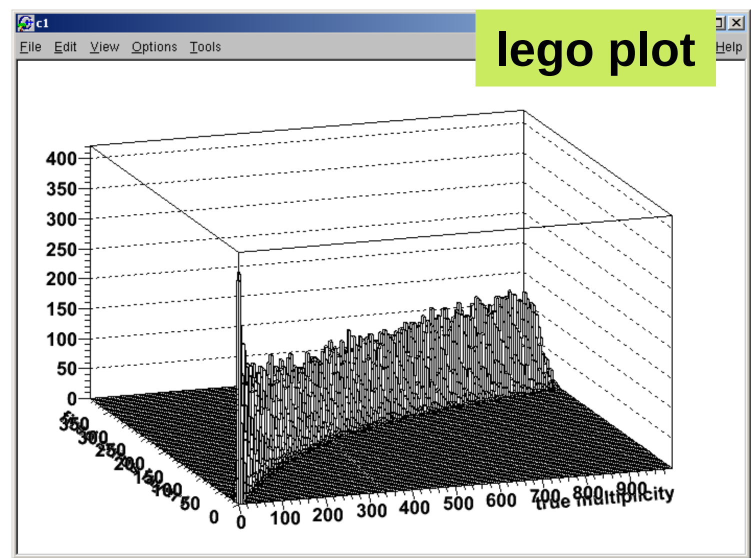
root [ ] h->Draw();

root [ ] h->Draw("LEGO");

root [ ] h->Draw("COLZ");







Files

- The class TFile allows to store any ROOT object on the disk
- Create a histogram like before with

```
root [ ] h = new TH1F("hist", "my hist; Bins; Entries", 10, 0, 10);
```

etc.

"hist" will be the name in the file

- Open a file for writing

```
root [ ] file = TFile::Open("file.root", "RECREATE");
```

- Write an object into the file

```
root [ ] h->Write();
```

- Close the file

```
root [ ] file->Close();
```

Files-II

- Open a file for writing

```
root [ ] file = TFile::Open("file.root");
```

- Read the object from the file

```
root [ ] hist->Draw();
```

(only works on the command line)

- In the macro read the object with

```
TH1F* h = 0;
File->GetObject ("hist", h);
```

- What else is in the file

```
root [ ] .ls
```

- Open a file when starting root

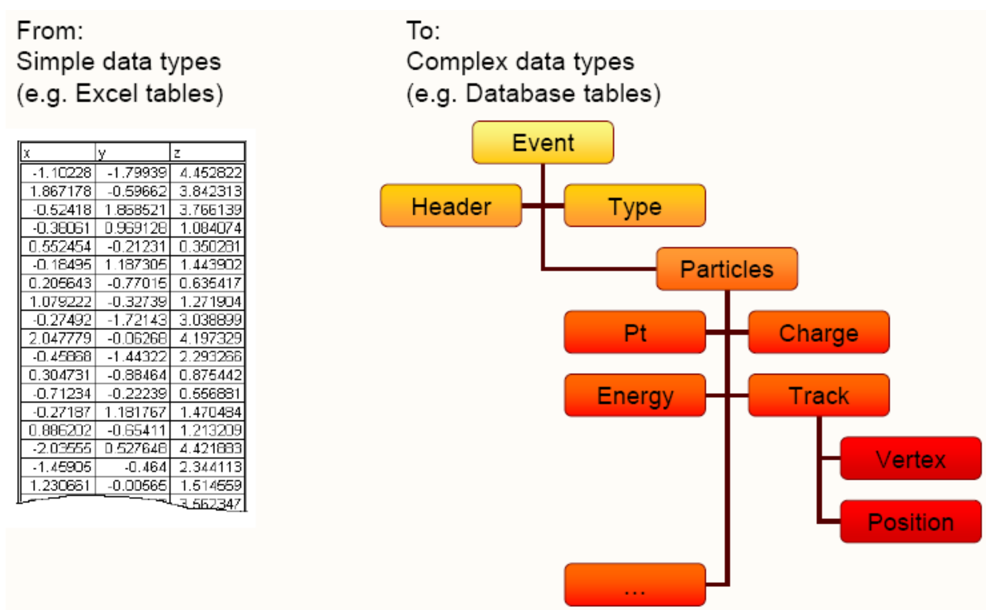
```
$ root file.root
```

Object ownership

After reading an object from a file don't close it! Otherwise your object is not in memory anymore

Trees

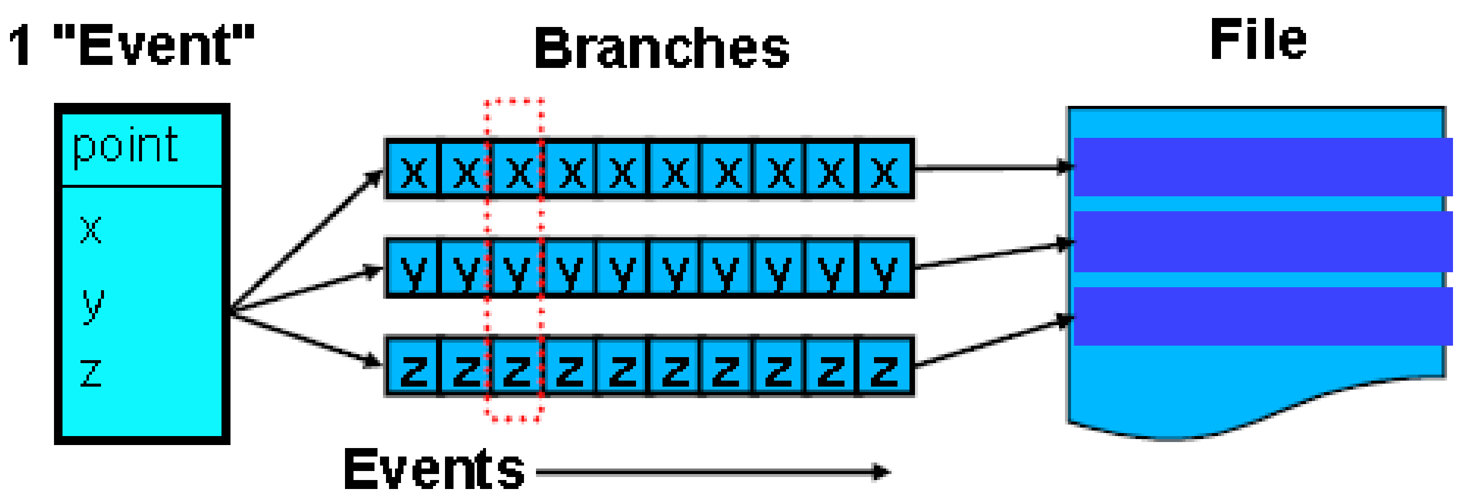
- A data type / structure which is convenient for HEP analysis
- Data bases have row-wise access
  – Can only access the full object (e.g. full event)
- Root Trees have column-wise access
  – Direct access to any event, any branch or any leaf even in the case of variable length structures



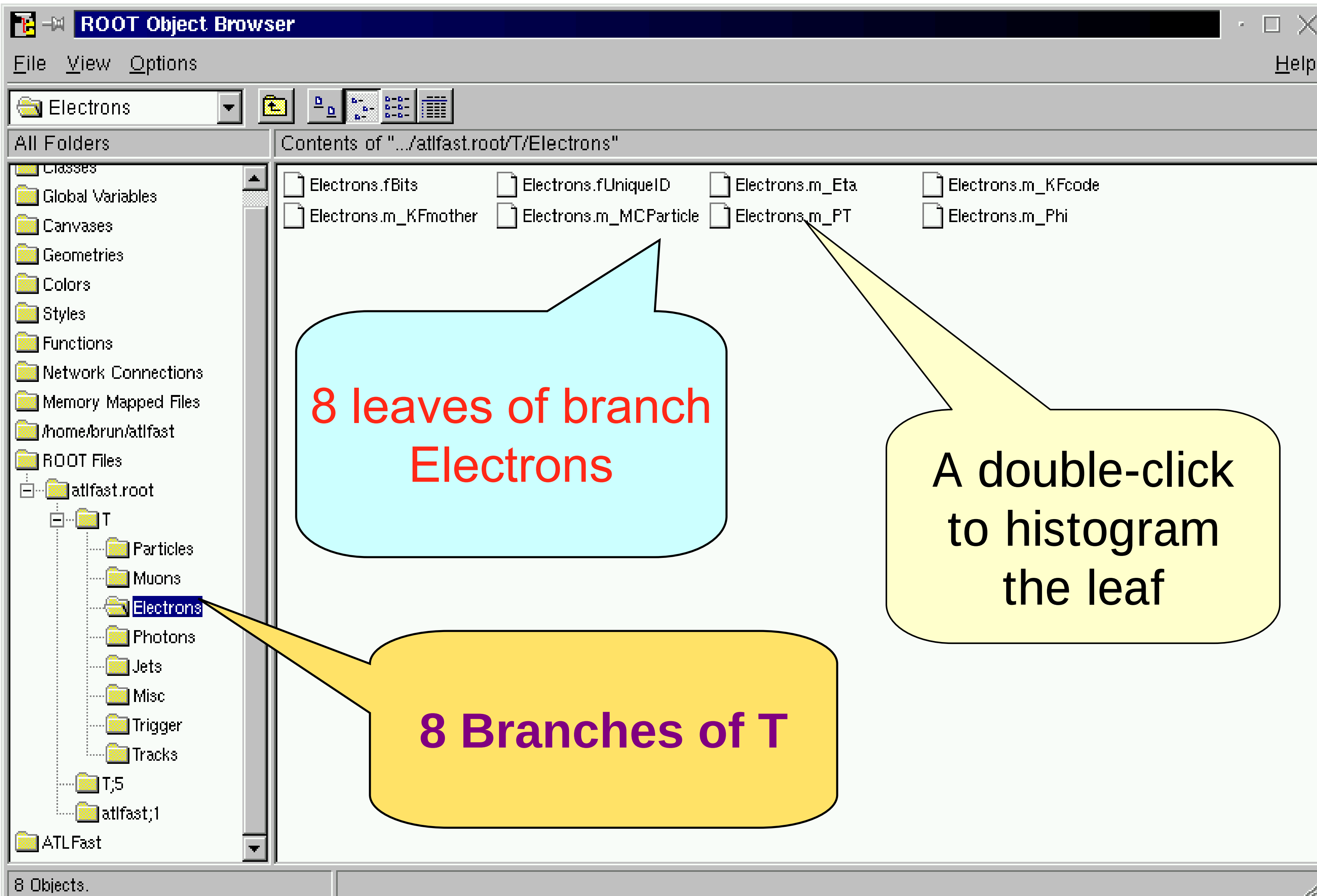
x	y	z
-1.10228	-1.79939	4.452822
1.867178	-0.59662	3.842313
-0.52418	1.868521	3.766139
-0.38061	0.969128	1.084074
0.552454	-0.21231	0.350281
-0.18495	1.187305	1.443902
0.205643	-0.77015	0.635417
1.079222	-0.32739	1.271904
-0.27492	-1.72143	3.038899
2.047779	-0.06268	4.197329
-0.45868	-1.44322	2.293266
0.304731	-0.88464	0.875442
-0.71234	-0.22239	0.556881
-0.27187	1.181767	1.470484
0.886202	-0.65411	1.213209
-2.03555	0.527648	4.421883
-1.45905	-0.464	2.344113
1.230661	-0.00565	1.514559
		3.562347

TTree

- The class TTree is the main container for data storage
  - It can store any class and basic types (e.g. Float_t)
  - When reading a tree, it is designed to access only a subset of the object attributes (e.g. only particle's energy) so that certain branches can be switched off → speed up of analysis when not all data is needed
- First example: the class TNtuple which is derived from TTree and contains only Float_t



Browsing a Tree



TNtuple

- Create a TNtuple

  root [ ] ntuple = new TNtuple("ntuple", "title", "x:y:z");

  – "ntuple" and “title" are the name and the title of the object

  – "x:y:z" reserves three variables named x, y, and z

- Fill it

  root [ ] ntuple->Fill(1, 1, 1);

- Get the contents

  root [ ] ntuple->GetEntries(); number of entries

  root [ ] ntuple->GetEntry(0); for the first entry

  root [ ] ntuple->Args()[1]; for y (0 for x, 2 for z)

  – These could be used in a loop to process all entries

- List the content

  root [ ] ntuple->Scan();

TNtuple-II

- Draw a histogram of the content
  - to draw only x

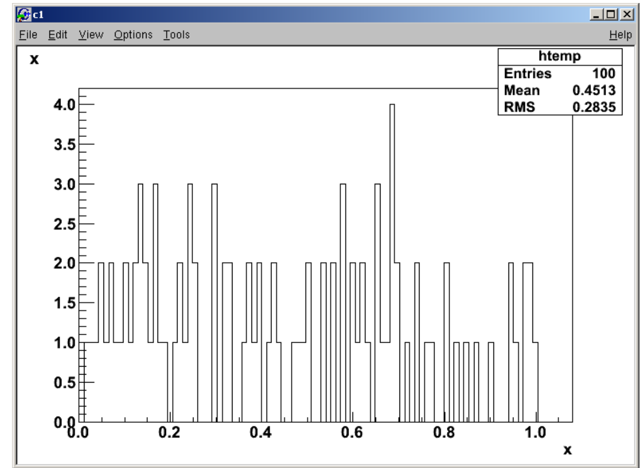
root [ ] ntuple->Draw("x");

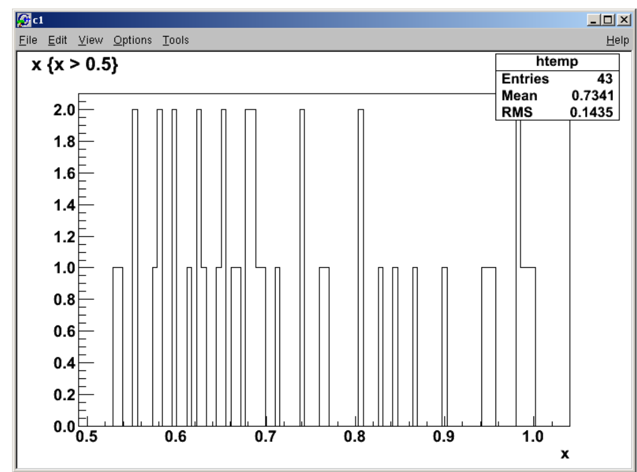
  - draw all x that fulfill x > 0.5

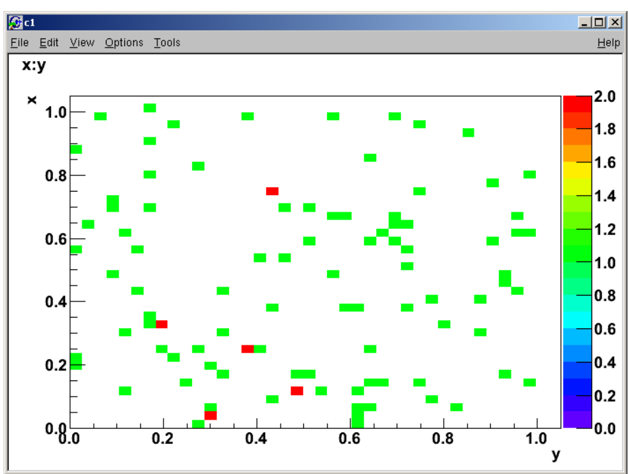
root [ ] ntuple->Draw("x", "x > 0.5");

  - to draw x vs. y in a 2d histogram

root [ ] ntuple->Draw("x:y ", "", "COLZ");



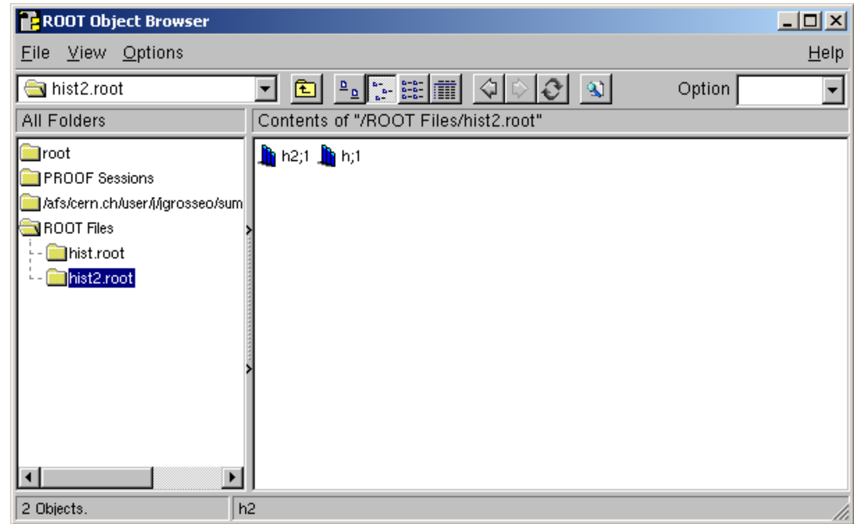


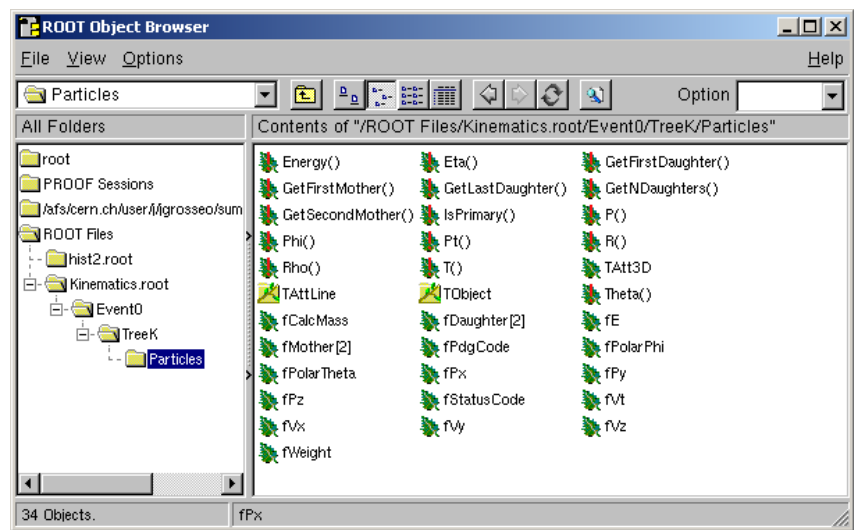


TBrowser

- The TBrowser can be used
  - to open files
  - navigate in them
  - to look at TTrees
  - draw a histogram
  - access a tree
  - change the standard style
  - plot a member
- Starting a TBrowser

  root [ ] TBrowser b;





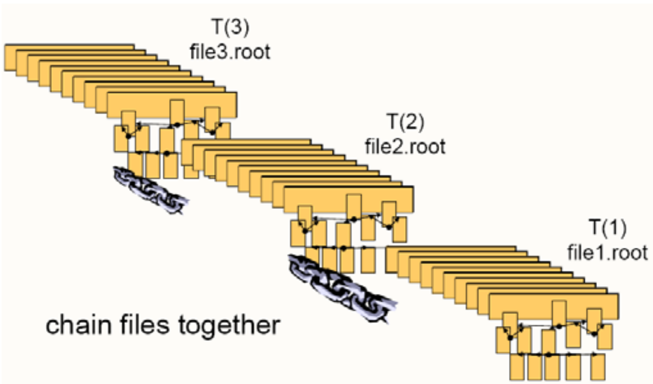
TChain : the Forest

- A chain is a list of trees (in several files)
- Normal TTree functions can be used

```
root [ ] chain= new TChain("tree");
root [ ] chain->Add ("tree1.root");
root [ ] chain->Add ("tree2.root");
root [ ] chain->Draw ("x");
```

Name of the tree in the files tree1.root and tree2.root

– The draw function iterates over both trees



Chain
Tree1 (File1)
Tree2 (File2)
Tree3 (File3)
Tree4 (File3)
Tree5 (File4)

Trees-II

- Connecting a class with the tree

```
root [ ] TParticle* particle = 0;
root [ ] tree->SetBranchAddress("Particles", &particle);
```

- Read an entry

```
root [ ] tree->GetEntry(0);
root [ ] particle->Print();
root [ ] tree->GetEntry(1);
root [ ] particle->Print();
```

The content of the instance TParticle is replaced with the current entry of the tree

  – These commands could be used in a loop to process all particles

```
root [5] particle->Print()
TParticle: pi0              p: -0.036864 -0.0
```

Understanding Errors

- Distinguish
  - Compiling error
    - ★ Syntax errors
    - ★ Missing declarations
  - Error while loading the library "dlopen error"
    - ★ Missing implementation of a declared function (much more subtle)
    - ★ Might even be in parent class
- Read error messages from top. Many other (weird) messages follow. Examples:
  - missing }
  - Missing include file
- Problems with macros? → Compile them to find errors

```
root [ ] .L macro.C
```

Basics of Debugging

- When there is a segmentation violation, you get the stack tree
  - It tells you where the crash happens
  - Find the relevant piece in the stack tree
    - ★ Start from top
    - ★ Few lines after "signal handler called"
    - ★ Most of the times it makes sense to look only at lines that refer to your own code
  - Compile with debug ("g") to see line numbers

Stack Tree

```
 *** Break *** segmentation violation
Using host libthread_db library "/lib/tls/libthread_db.so.1".
Attaching to program: /proc/23893/exe, process 23893
[Thread debugging using libthread_db enabled]
[New Thread -1208858944 (LWP 23893)]
0x0077c7a2 in _dl_sysinfo_int80 () from /lib/ld-linux.so.2
#1  0x002b34b3 in __waitpid_nocancel () from /lib/tls/libc.so.6
#2  0x0025c779 in do_system () from /lib/tls/libc.so.6
#3  0x0022198d in system () from /lib/tls/libpthread.so.0
#5  0x009db83e in TUnixSystem::StackTrace (this=0x9daa440) at core/unix/src/TUnixSystem.cxx:2132
#6  0x009d962d in TUnixSystem::DispatchSignals (this=0x9daa440, sig=kSigSegmentationViolation) at core/unix/src/TUnixSys
#7  0x009d745d in SigHandler (sig=kSigSegmentationViolation) at core/unix/src/TUnixSystem.cxx:350
#8  0x009de7aa in sighandler (sig=11) at core/unix/src/TUnixSystem.cxx:3368
#9  <signal handler called>
#10 0x003effd8 in TSummerStudent::SomeFunction (this=0xa0154b0) at /home/shuttle/Fiete/./TSummerStudent_debug.C:14
#11 0x003ee355 in G__TSummerStudent_debug_C_ACLiC_dict_2564_0_3 (result7=0xbffe0420, funcname=0xa0153f8 "\001", libp=0xb
    at /home/shuttle/Fiete/./TSummerStudent_debug_C_ACLiC_dict.cxx:186
#12 0x00ed8dbf in Cint::G__ExceptionWrapper (funcp=0x3ee32e <G__TSummerStudent_debug_C_ACLiC_dict_2564_0_3>, result7=0xb
    hash=0) at cint/cint/src/Api.cxx:384
#13 0x00f81786 in G__execute_call (result7=0xbffe0420, libp=0xbffda5a0, ifunc=0xa0153f8, ifn=0) at cint/cint/src/newlink
#14 0x00f81ea6 in G__call_cppfunc (result7=0xbffe0420, libp=0xbffda5a0, ifunc=0xa0153f8, ifn=0) at cint/cint/src/newlink
#15 0x00f6295a in G__interpret_func (result7=0xbffe0420, funcname=0xbffe0020 "SomeFunction", libp=0xbffda5a0, hash=1242,
    at cint/cint/src/ifunc.cxx:5277
#16 0x00f4907c in G__getfunction (item=0xbffe3263 "SomeFunction()", known3=0xbffe267c, memfunc_flag=1) at cint/cint/src/
#17 0x0103b145 in G__getstructmem (store_var_type=112, varname=0xbffe0670 "�/5", membername=0xbffe3263 "SomeFunction()",
    varglobal=0x10d9ea0, objptr=2) at cint/cint/src/var.cxx:6691
#18 0x0102f234 in G__getvariable (item=0xbffe3260 "s->SomeFunction()", known=0xbffe267c, varglobal=0x10d9ea0, varlocal=0
#19 0x00f3ccc9 in G__getitem (item=0xbffe3260 "s->SomeFunction()") at cint/cint/src/expr.cxx:1884
#20 0x00f3b338 in G__getexpr (expression=0xbffe4b50 "s->SomeFunction()") at cint/cint/src/expr.cxx:1470
```

Basics of Debugging-II

- Reproduce the problem in the debugger
- Most linux systems include gdb GNU debugger

  $ gdb root.exe (gdb root does not work)

  – Parameters to root have to be passed with

  $ gdb –args root.exe macro.C

  – On the gdb prompt, start the program : (gdb) run
- You will see the line where the crash happened
- Basic commands

  – bt = backtrace, gives the stack

  – up, down to navigate in the stack → go to the first frame with your code

  – p <var> → prints the variable <var> (of your code e.g. particle)

  – quit to exit

Resources

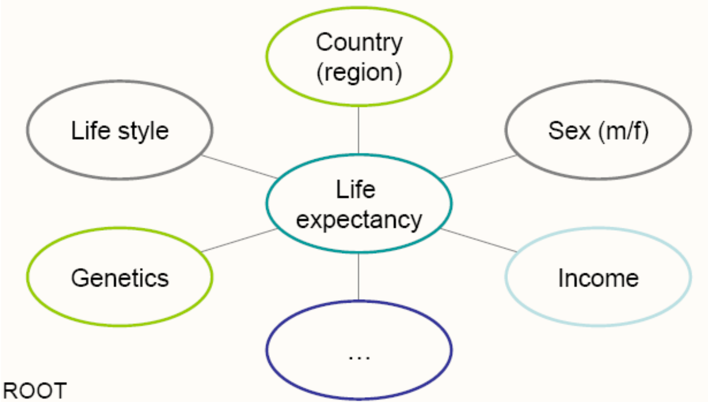
- Main ROOT page
  - http://root.cern.ch
- Class Reference Guide
  - http://root.cern.ch/root/html
- C++ tutorial
  - http://www.cplusplus.com/doc/tutorial

BACKUP SLIDES

Why Simulated Data?

- Generators acts like accelerators (LHC,LEP,TEVATRON)
- Allow theoretical and experimental studies of complex multi-particle physics
- Vehicle of ideology to disseminate ideas from theorists to experimentalists
- Predict the event rates and topology (Kinematics of particles resulted from collisions)
- To trace back the history of end products need
- Simulate possible backgrounds
- Study detector requirements

Multi-Variate Analysis: TMVA



- What is a multi-variate problem
- Example: A person's life expectancy
- Depends on many variables
- TMVA
  - A Framework offering a collection of data mining tools, e.g. NN (Neural Network), GA (Genetic Algorithm), …
  - In HEP mostly two class problems – signal (S) and background (B) Physics processes
  - Finding physics objects
  - Detector readout

Saving Data

- Streaming
- Reflection
- TFile
- Evolution scheme

Saving Objects

- Cannot do in C++:
  - **TNamed* o; TNamed* p;**
  - **o = new TNamed("name", "title");**
  - **std::write("file.bin", "obj1", o);**
  - **p = std::read("file.bin", "obj1");**
  - **p->GetName();**
- E.g. LHC experiments use C++ to manage data
- Need to write C++ objects and read them back
- std::cout not an option: 15PetaBytes / year of
- processed data (i.e. data that will be read)

Saving Types

- What's needed?
  - **TNamed* o;**
  - **o = new TNamed("name", "title");**
  - **std::write("file.bin", "obj1", o);**
- Store *data members of TNamed; need to know:*
- 1) type of object
- 2) data members for the type
- 3) where data members are in memory
- 4) read their values from memory, write to disk

Serialization

- *A process which Store data members of Tnamed in following steps*
  - type of object: run-time-type-information RTTI
  - data members for the type: reflection
  - where data members are in memory: introspection
  - read their values from memory, write to disk: raw I/O

Reflection

- Need type description (aka *reflection)*
  - types, sizes, members
- TMyClass is a class.
- Members:
  - "fFloat", type float, size 4 bytes
  - "fLong", type Long64_t, size 8 bytes
  - **class TMyClass {float fFloat; Long64_t fLong; };**

Platform Data Types

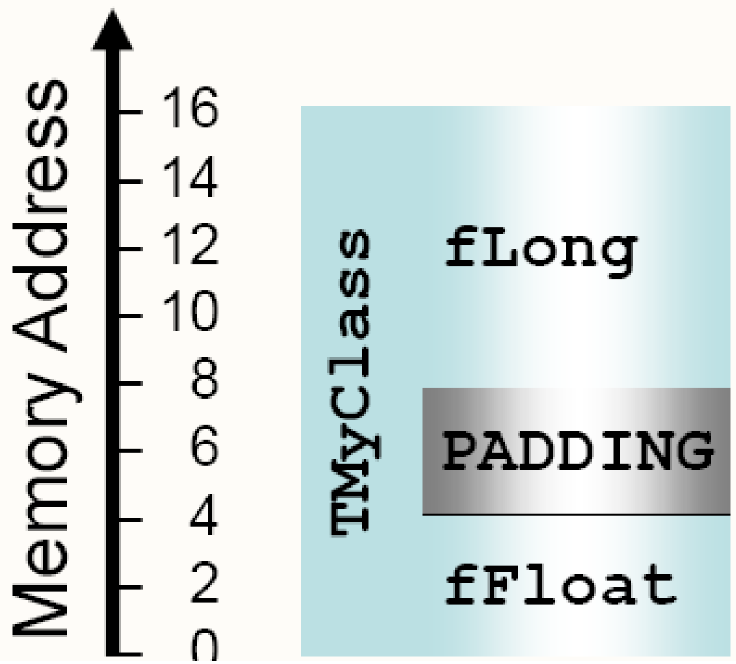
- Fundamental data types (int, long,…):
- size is platform dependent
- Store "long" on 64bit platform, writing 8 bytes:
- 00, 00, 00, 00, 00, 00, 00, 42
- Read on 32bit platform, "long" only 4 bytes:
- 00, 00, 00, 00
- Data loss, data corruption!

How Reflection is achieved

Need type description (platform dependent)

1. types, sizes, members
2. offsets in memory

```
class TMyClass {
  float fFloat;
  Long64_t fLong;
};
```



"fFloat" is at offset 0

"fLong" is at offset 8

Saving Objects Continued

- Given a TFile:
  - **TFile* f = new TFile("file.root", "RECREATE");**
- Write an object deriving from TObject:
  - **object->Write("optionalName")**
- "optionalName" or **TObject::GetName()**
- Write any object (with dictionary):
  - **f->WriteObject(object, "name");**

TFile Properties

- ROOT stores objects in TFiles:
  - **TFile* f = new TFile("file.root", "NEW");**
- TFile behaves like file system:
  - **f->mkdir("dir");**
- TFile has a current directory:
  - **f->cd("dir");**
- TFile compresses data ("zip"):
  - **f->GetCompressionFactor()**
  - **2.61442160606384277e00**

scope for hists, graphs, trees

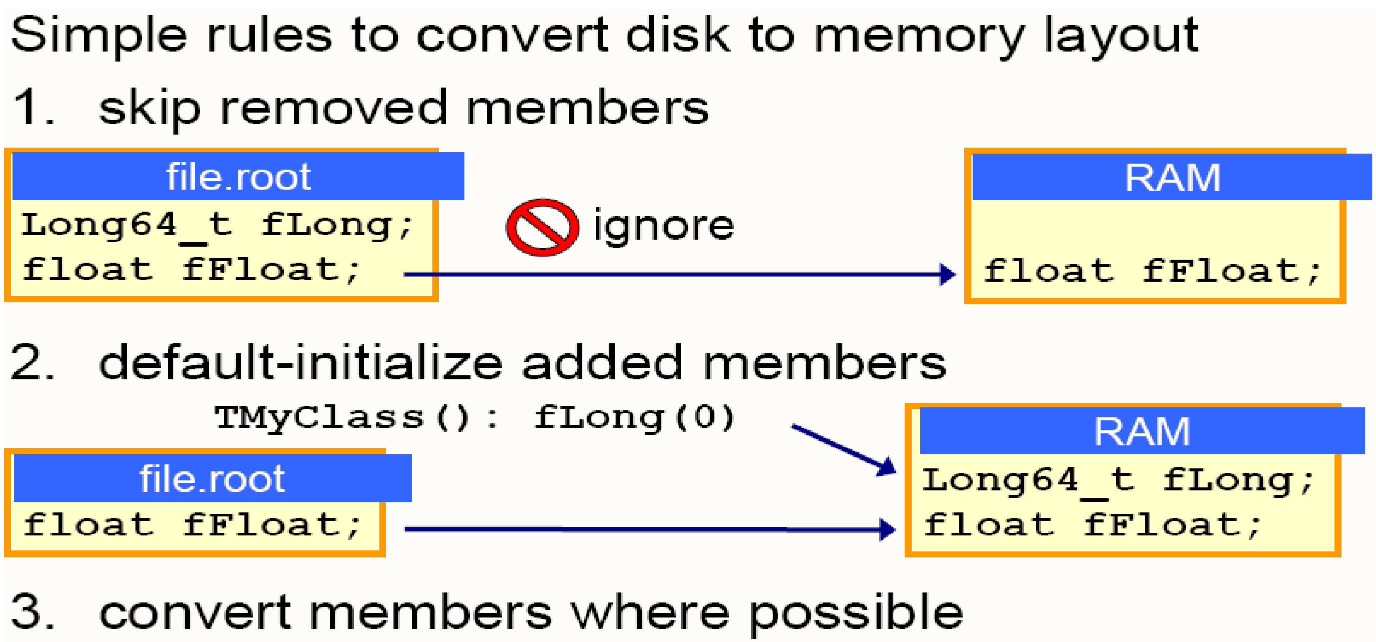
- TFile owns histograms, graphs, trees
  - **TFile* f = new TFile("myfile.root");**
  - **TH1F* h = new TH1F("h","h",10,0.,1.);**
  - **h->Write();**
  - **TCanvas* c = new TCanvas();**
  - **c->Write();**
  - **delete f;**
- h automatically deleted: owned by file.
- c still there. → *names unique!*
- TFile acts like a scope for hists, graphs, trees!

Ownership And TFiles

- Separate TFile and histograms
  - **TFile* f = new TFile("myfile.root");**
  - **TH1F* h = 0;**
  - **TH1::AddDirectory(kFALSE);**
  - **f->GetObject("h", h);**
  - **h->Draw();**
  - **delete f;**
- … and h will stay around.

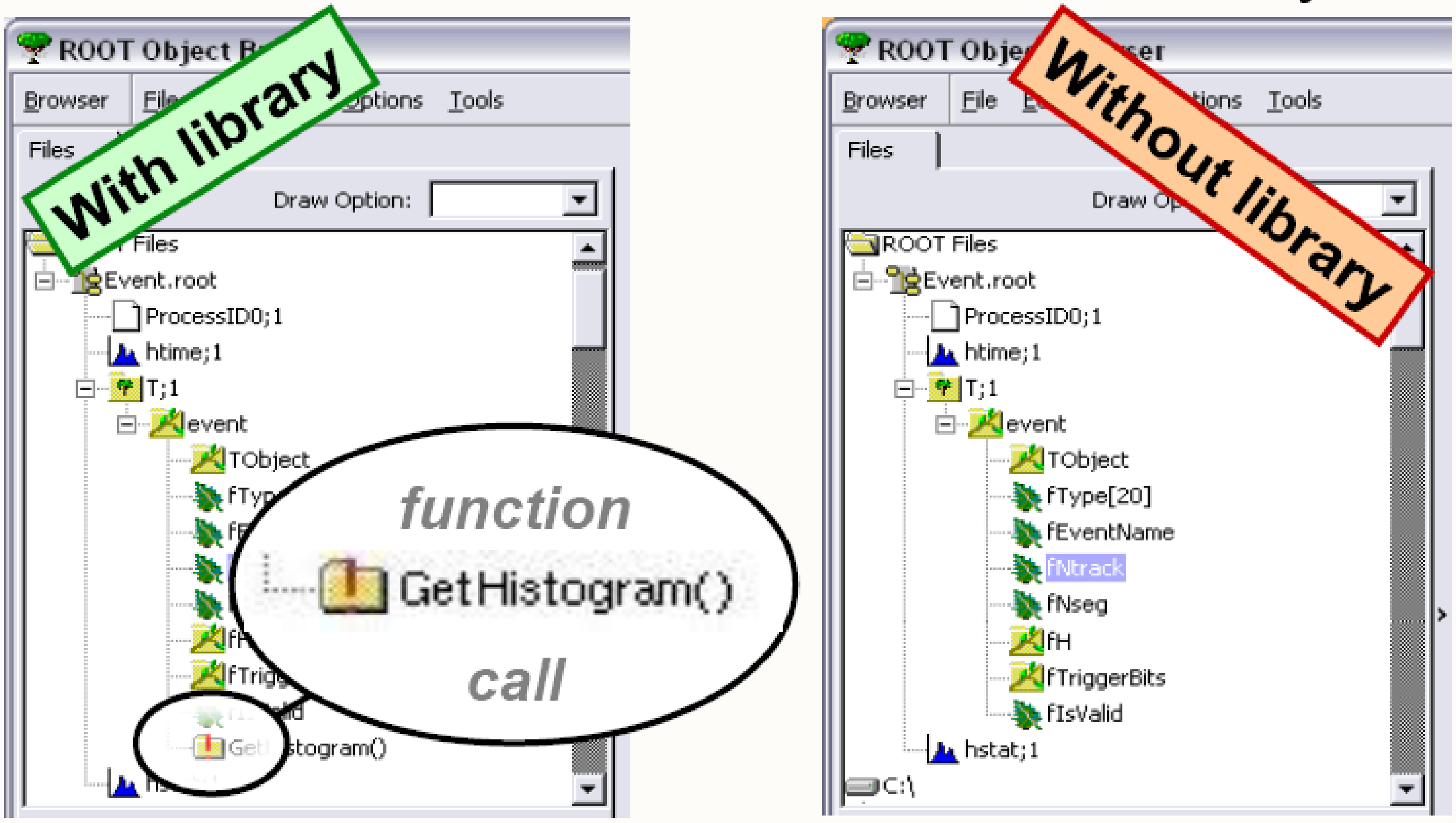
Evolution Scheme

- Changing Class can be a Problem:
  - **class TMyClass {double fFloat;Long64_t Long;};**
- nconsistent reflection data, mismatch in memory, on disk
- Objects written with old version cannot be read
- *Need to store reflection with data to detect!*



Reading Files

Files store reflection and data: need no library!



Powers of ROOT I/O

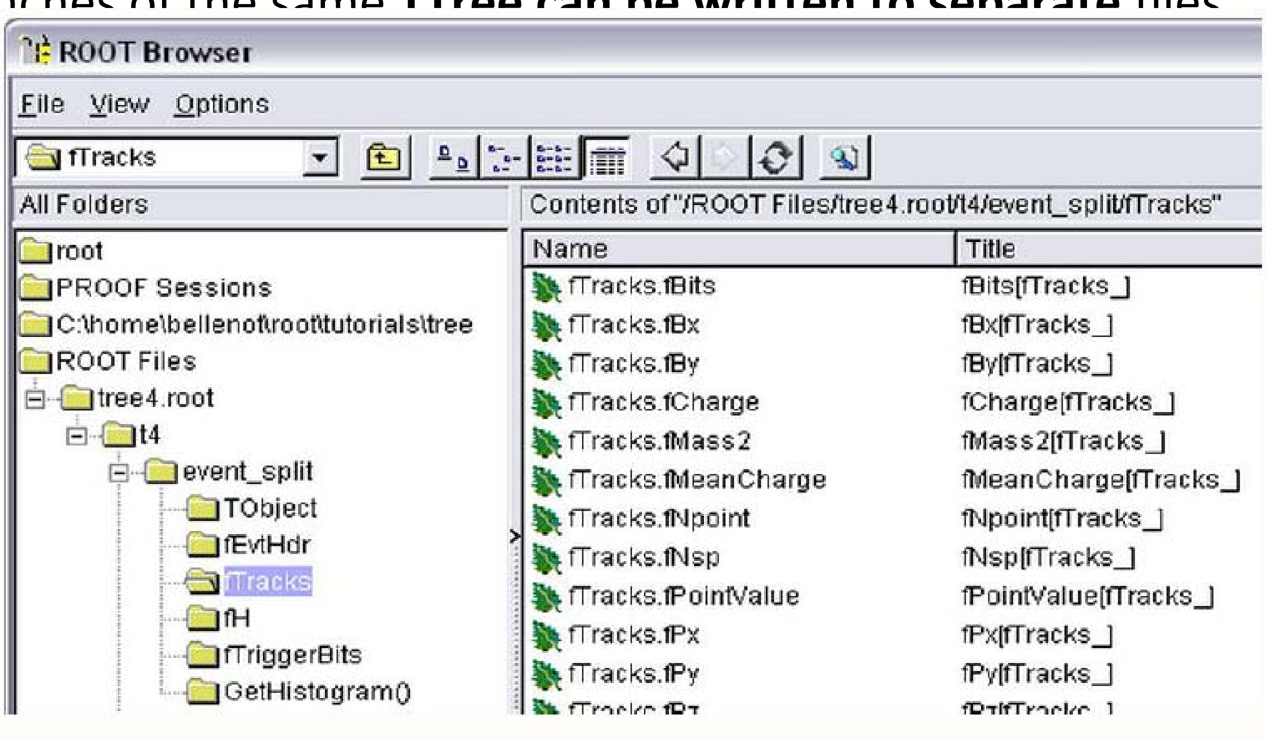
- Can even open
  - **TFile::Open("http://cern.ch/file.root")**
- including read-what-you-need!
- Nice viewer for TFile: **new TBrowser**
- Combine contents of TFiles with
  - **$ROOTSYS/bin/hadd**

Trees

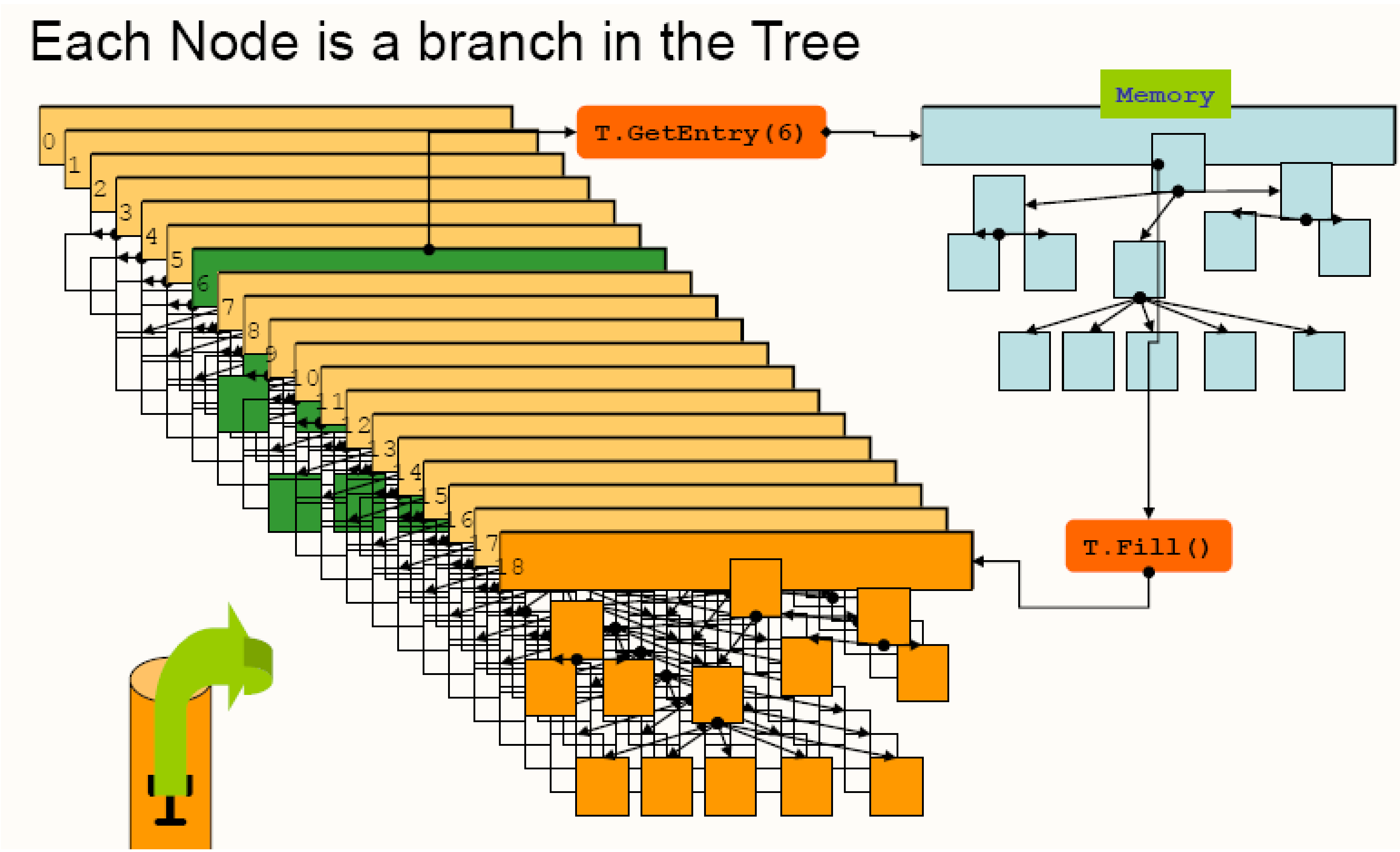
- Databases have row wise access
  - Can only access the full object (e.g. full event)
- ROOT trees have column wise access
  - Direct access to any event, any branch or any leaf even in the case of variable length structures
  - Designed to access only a subset of the object attributes (e.g. only particles' energy)
- object.Write() convenient for simple objects like histograms, inappropriate for saving collections of events containing complex objects
  - Reading a collection: read all elements (all events)
  - With trees: only one element in memory, or even only a part of it (less I/O)

Tree structure

- Branches: directories
  - Leaves: data containers
  - Can read a subset of all branches – speeds up
- considerably the data analysis processes
  - Branches of the same **TTree can be written to separate** files



Memory ←→ Tree



Each Node is a branch in the Tree
T.GetEntry(6)
Memory
T.Fill()
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18

Five Steps to Build a Tree

- Steps:

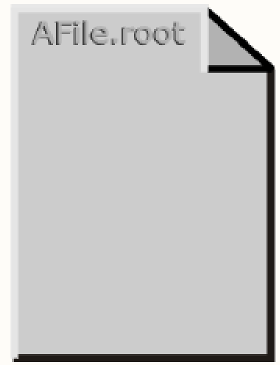
1. Create a TFile
2. Create a TTree
3. Add TBranch to the TTree
4. Fill the tree
5. Write the file

Example to Build/Write a Tree

```
void WriteTree()
{
Event *myEvent = new Event();
TFile f("AFile.root");
TTree *t = new TTree("myTree","A Tree");
t->Branch("EventBranch", &myEvent);
for (int e=0;e<100000;++e) {
myEvent->Generate(); // hypothetical
t->Fill();
}
t->Write();
}
```

Step 1: Create TFile
Step2: Create TTree

Trees can be huge → need file for swapping filled entries




```
TFile *hfile = new TFile("AFile.root");
```

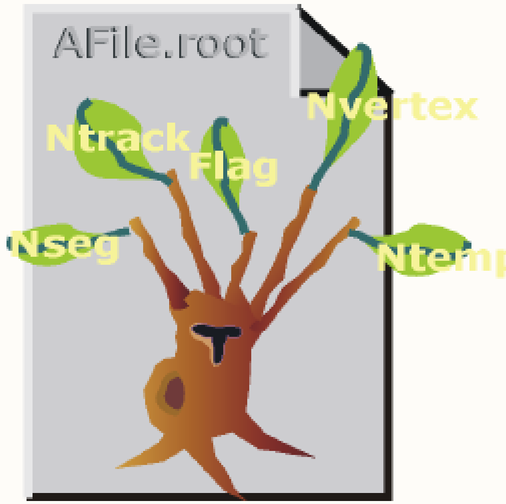
The TTree constructor:

- Tree name (e.g. "myTree")
- Tree title


```
TTree *tree = new TTree("myTree","A Tree");
```

Step3: Add Branch

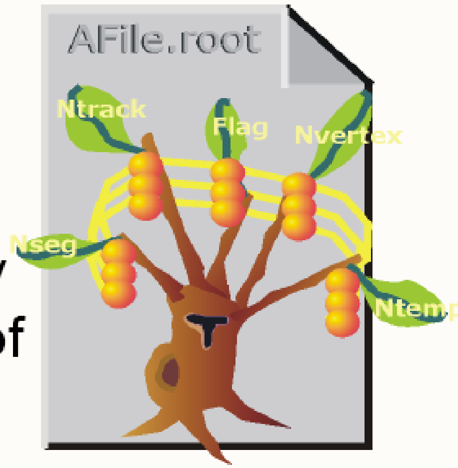
- Branch name
- Pointer to the object




```
Event *myEvent = new Event();
myTree->Branch("eBranch", &myEvent);
```

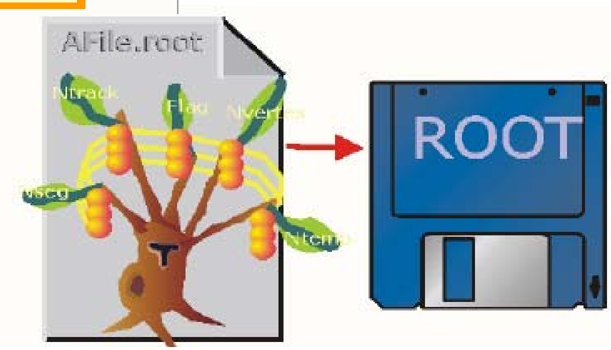
Step 4: Fill Tree
Step5: Write Tree to File

- Create a for loop
- Assign values to the object contained in each branch
- TTree::Fill() creates a new entry in the tree: snapshot of values of branches' objects




```
for (int e=0;e<100000;++e) {
   myEvent->Generate(e); // fill event
   myTree->Fill();       // fill the tree
}
```

```
myTree->Write();
```



Example to Read a Tree

```
void ReadTree()
{
Event *myEvent = 0;
TFile f("AFile.root");
TTree *myTree = (TTree*)f->Get("myTree");
myTree->SetBranchAddress("EventBranch",
&myEvent);
for (int e=0;e<100000;++e) {
myTree->GetEntry(e);
myEvent->Analyze();
}
}
```

Read a Tree Continued

Example:

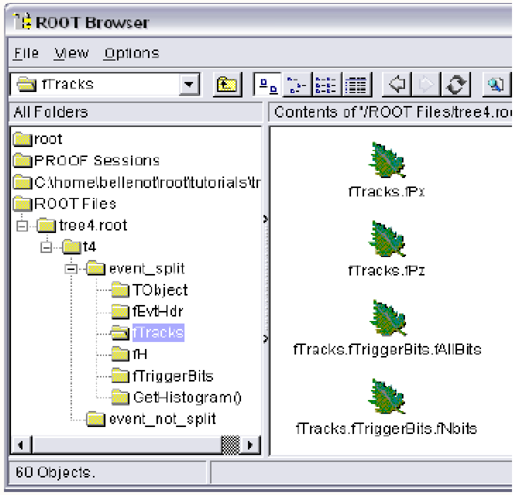
1. Open the Tfile

```
TFile f("AFile.root")
```

2. Get the TTree

```
TTree *myTree = 0;
f.GetObject("myTree",myTree)
```

OR



3. Create a variable pointing to the data

```
root [] Event *myEvent = 0;
```

4. Associate a branch with the variable:

```
root [] myTree->SetBranchAddress("eBranch", &myEvent);
```

5. Read one entry in the TTree

```
root [] myTree->GetEntry(0)
root [] myEvent->GetTracks()->First()->Dump()
==> Dumping object at: 0x0763aad0, name=Track, class=Track
fPx             0.651241    X component of the momentum
fPy             1.02466     Y component of the momentum
fPz             1.2141      Z component of the momentum
[...]
```

Branch Access Selection

- Use TTree::SetBranchStatus() to activate only the
- branches holding wanted variables.
  - Speed up considerably the reading phase
  - **TClonesArray* myMuons = 0;**
  - **// disable all branches**
  - **myTree->SetBranchStatus("*", 0);**
  - **// re-enable the "muon" branches**
  - **myTree->SetBranchStatus("muon*", 1);**
  - **myTree->SetBranchAddress("muon", &myMuons);**
  - **// now read (access) only the "muon" branches**
  - **myTree->GetEntry(0);**

Looking at the Tree

TTree::Print() shows the data layout

```
root [] TFile f("AFile.root")
root [] myTree->Print();
******************************************************************************
*Tree    :myTree    : A ROOT tree                                            *
*Entries :       10 : Total =           867935 bytes  File  Size =     390138 *
*        :          : Tree compression factor =   2.72                       *
******************************************************************************
*Branch  :eBranch                                                        *
*Entries :       10 : BranchElement (see below)                              *
*............................................................................*
*Br    0 :fUniqueID :                                                        *
*Entries :       10 : Total  Size=        698 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      64000 bytes  Compression=   1.00     *
*............................................................................*
…
…
```

TTree::Scan("leaf:leaf:….") shows the values

```
root [] myTree->Scan("fNseg:fNtrack"); > scan.txt

root [] myTree->Scan("fEvtHdr.fDate:fNtrack:fPx:fPy","",
                     "colsize=13 precision=3 col=13:7::15.10");

*********************************************************************
* Row * Instance * fEvtHdr.fDate * fNtrack *         fPx *            fPy *
*********************************************************************
*   0 *        0 *        960312 *     594 *        2.07 *    1.459911346 *
*   0 *        1 *        960312 *     594 *       0.903 *  -0.4093382061 *
*   0 *        2 *        960312 *     594 *       0.696 *   0.3913401663 *
*   0 *        3 *        960312 *     594 *      -0.638 *    1.244356871 *
*   0 *        4 *        960312 *     594 *      -0.556 *  -0.7361358404 *
*   0 *        5 *        960312 *     594 *       -1.57 *  -0.3049036264 *
*   0 *        6 *        960312 *     594 *      0.0425 *   -1.006743073 *
*   0 *        7 *        960312 *     594 *        -0.6 *   -1.895804524 *
```

Looking at the Tree Continued

TTree::Show(entry_number) shows the values for one entry

```
 root [] myTree->Show(0);
======> EVENT:0
 eBranch          = NULL
 fUniqueID        = 0
 fBits            = 50331648
 [...]
 fNtrack          = 594
 fNseg            = 5964
 [...]
 fEvtHdr.fRun     = 200
 [...]
 fTracks.fPx      = 2.066806, 0.903484, 0.695610, -0.637773,...
 fTracks.fPy      = 1.459911, -0.409338, 0.391340, 1.244357,...
```

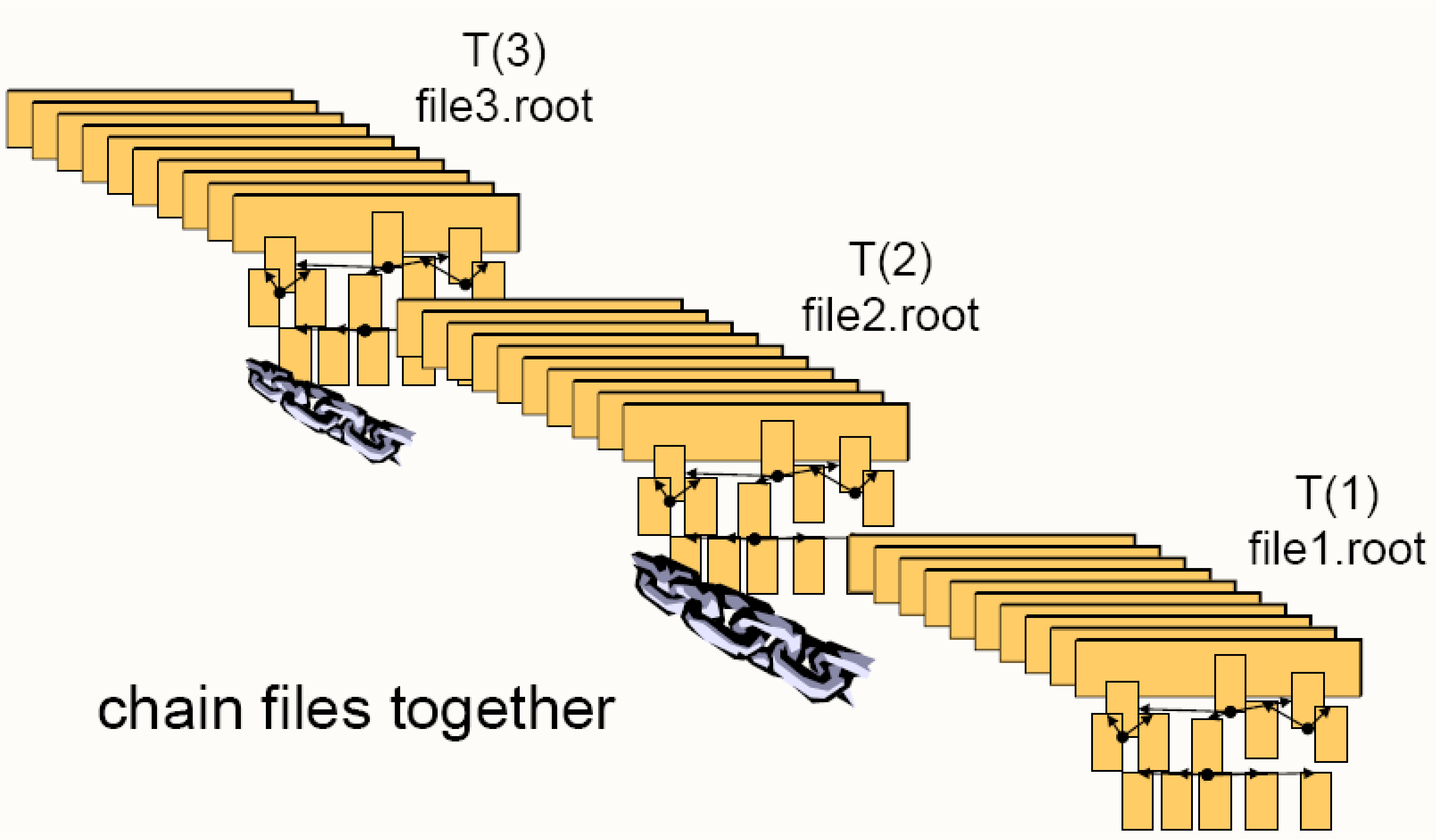
TTree Selection Syntax

  - **MyTree->Scan();**
- Prints the first 8 variables of the tree.
  - **MyTree->Scan("*");**
- Prints all the variables of the tree.
- Select specific variables:
  - **MyTree->Scan("var1:var2:var3");**
- Prints the values of var1, var2 and var3.
- A selection can be applied in the second argument:
  - **MyTree->Scan("var1:var2:var3", "var1>0");**
- Prints the values of var1, var2 and var3 for the entries where var1 is greater than 0
- Use the same syntax for TTree::Draw()

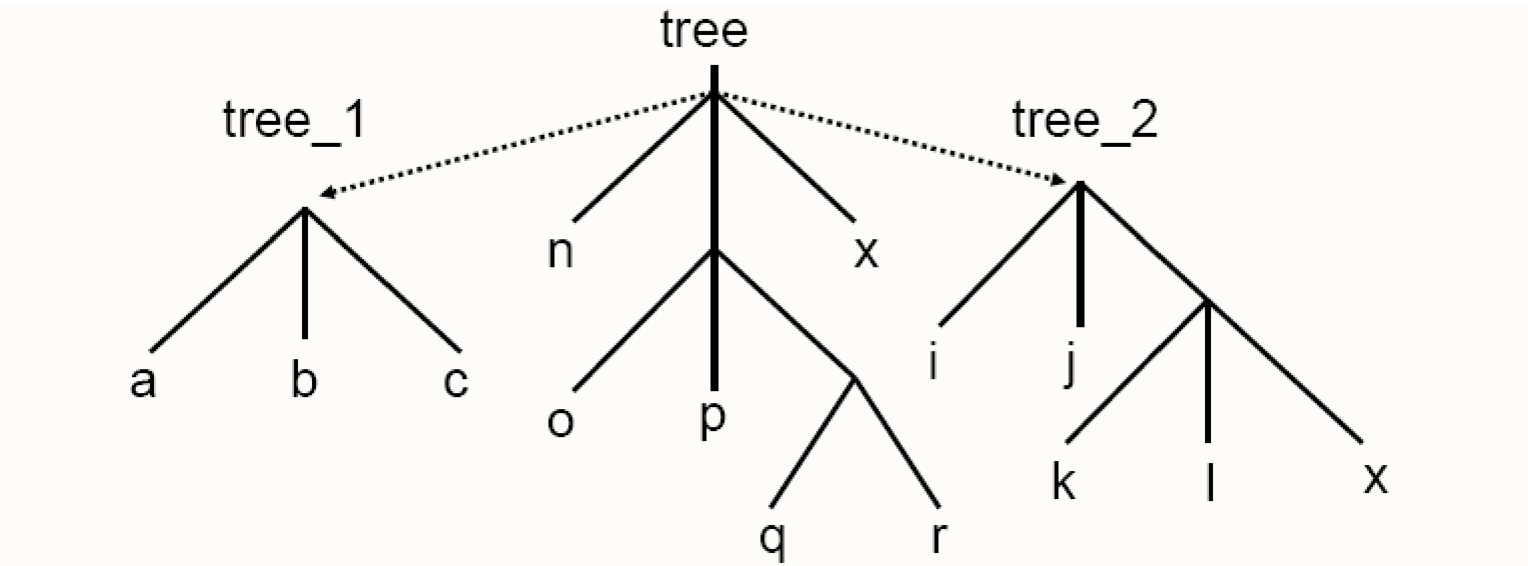
TChain: the Forest

- Collection of TTrees: list of ROOT files containing the
- same tree
- • Same semantics as TTree
- As an example, assume we have three files called
- file1.root, file2.root, file3.root. Each contains tree called
- "T". Create a chain:
  - **TChain chain("T");** ***// argument: tree name***
  - **chain.Add("file1.root");**
  - **chain.Add("file2.root");**
  - **chain.Add("file3.root");**
- Now we can use the TChain like a TTree!

TChain


T(3)
file3.root
T(2)
file2.root
T(1)
file1.root
chain files together

Tree Friends

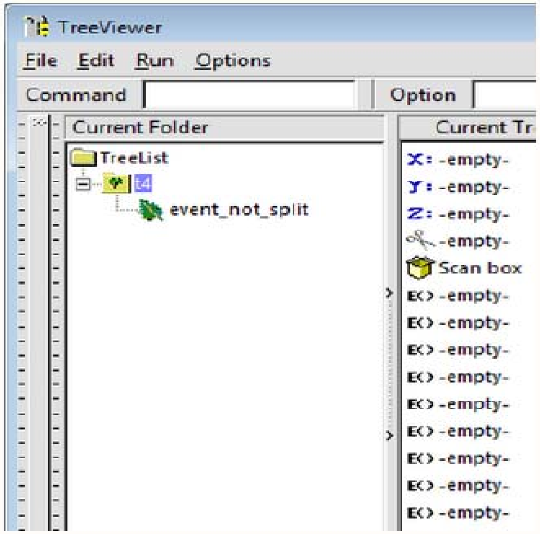



```
TFile f1("tree1.root");
tree.AddFriend("tree_1", "tree2.root")
tree.AddFriend("tree_2", "tree3.root");
tree.Draw("x:a", "k<c");
tree.Draw("x:tree_2.x", "sqrt(p)<b");
```

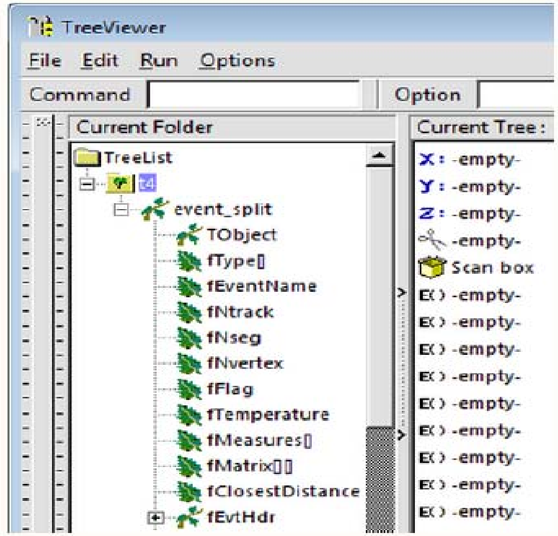
Tree Friends Continued

- Trees are designed to be read only
  - Often, people want to add branches to existing
- trees and write their data into it
  - Using tree friends is the solution:
  - Create a new file holing the new tree
  - Create a new Tree holding the branches for the users data
  - Fill the tree/branches with users data
  - Add this new file/tree as friend of the original tree

Splitting



Split level = 0

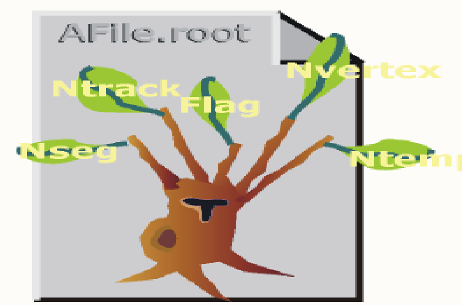


Split level = 99

Default

Setting the split level (default = 99)

Split level = 0



Split level = 99

```
tree->Branch("EvBr", &event, 64000, 0 );
```

Splitting Continued

- Creates one branch per member – recursively
  - Allows to browse objects that are stored in trees,
- even without their library
  - Makes same members consecutive, e.g. for object with position in X, Y, Z, and energy E, all X are consecutive, then come Y, then Z, then E. A lot higher zip efficiency!
  - Fine grained branches allow fain-grained I/O -
- read only members that are needed
  - Supports STL containers too, even vector<T*>!
- A split branch is:
  - Faster to read – if you only want a subset of data members
  - Slower to write due to the large number of branches
  - Higher compressed