ROOT Tutorial at Desy C++ School 2013

15 November 2013

Dr Lorenzo Moneta
CERN PH-SFT
CH-1211 Geneva 23
sftweb.cern.ch
root.cern.ch

Outline

- Introduction to ROOT
- ROOT I/O
- Trees in ROOT
  - how to create and fill trees
  - how to read and analyze Trees
    - using TTree::Draw, C++ code and TSelector class
- Parallel Tree data analysis using PROOF

Introduction to ROOT

- What is ROOT ?
- Overview of ROOT functionality

Dr Lorenzo Moneta
CERN PH-SFT
CH-1211 Geneva 23
sftweb.cern.ch
root.cern.ch

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: histogramming, fitting, minimization and multi-variate analysis algorithms
  - scientific visualization: 2D and 3D graphics, Postscript, PDF, LateX
  - geometrical modeler
  - PROOF parallel query engine

- An Open Source Project

ROOT: An Open Source Project

- Started in 1995 by *R. Brun* and *F. Rademakers.*

ROOT: An Open Source Project

- Started in 1995 by *R. Brun* and *F. Rademakers.*
- Available (including the source) under GNU LGPL.

ROOT: An Open Source Project

- Started in 1995 by *R. Brun* and *F. Rademakers.*
- Available (including the source) under GNU LGPL.
- 7 full time developers at CERN, plus 2 at Fermilab (Chicago).

ROOT: An Open Source Project

- Started in 1995 by *R. Brun* and *F. Rademakers.*
- Available (including the source) under GNU LGPL.
- 7 full time developers at CERN, plus 2 at Fermilab (Chicago).
- Many contributors from high energy physics experiments which uses ROOT as base for their software frameworks.

ROOT: An Open Source Project

- Started in 1995 by *R. Brun* and *F. Rademakers.*
- Available (including the source) under GNU LGPL.
- 7 full time developers at CERN, plus 2 at Fermilab (Chicago).
- Many contributors from high energy physics experiments which uses ROOT as base for their software frameworks.
- Large number of part-time developers.

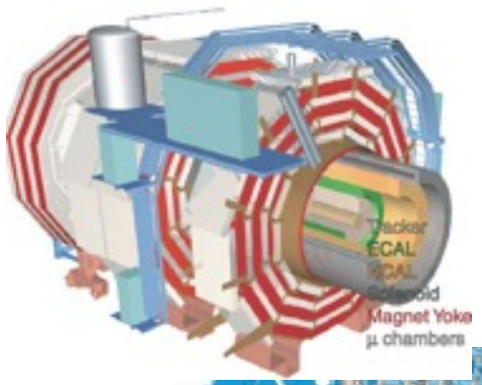
ROOT: An Open Source Project

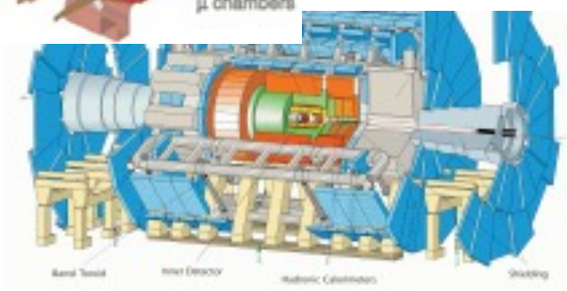
- Started in 1995 by *R. Brun* and *F. Rademakers.*
- Available (including the source) under GNU LGPL.
- 7 full time developers at CERN, plus 2 at Fermilab (Chicago).
- Many contributors from high energy physics experiments which uses ROOT as base for their software frameworks.
- Large number of part-time developers.
- Several thousands of users giving feedback and a very long list of small contributions.

Why ROOT ?

- The analysis of data coming from LHC experiments (and also other experiments) requires a powerful and general toolkit
  - Visualisation
  - Statistical studies
  - Data reduction
  - Multivariate techniques
- A scalable and reliable persistency method is needed to write the data on disks and tapes.

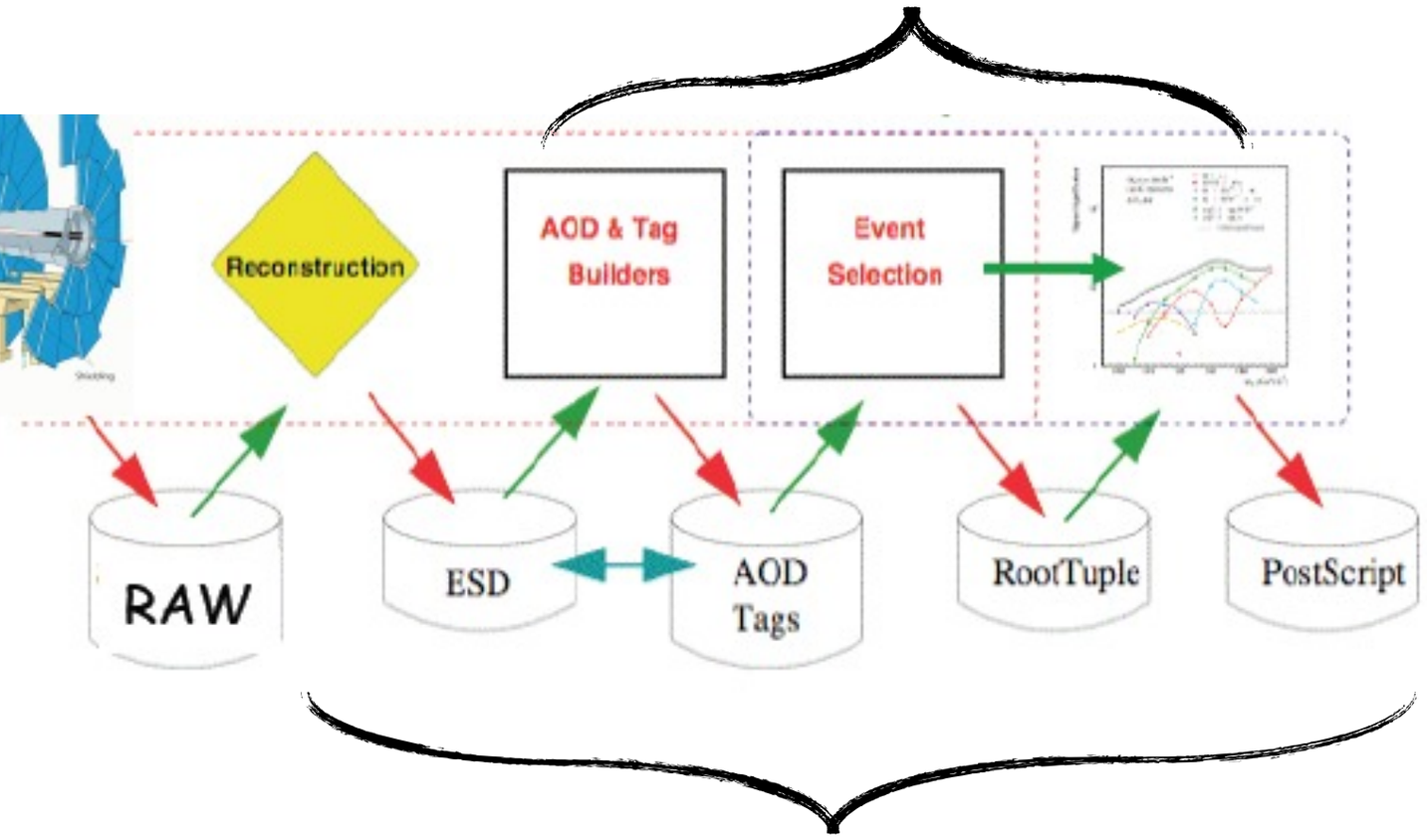
ROOT Application Domains





General Framework

Data Analysis & Visualization

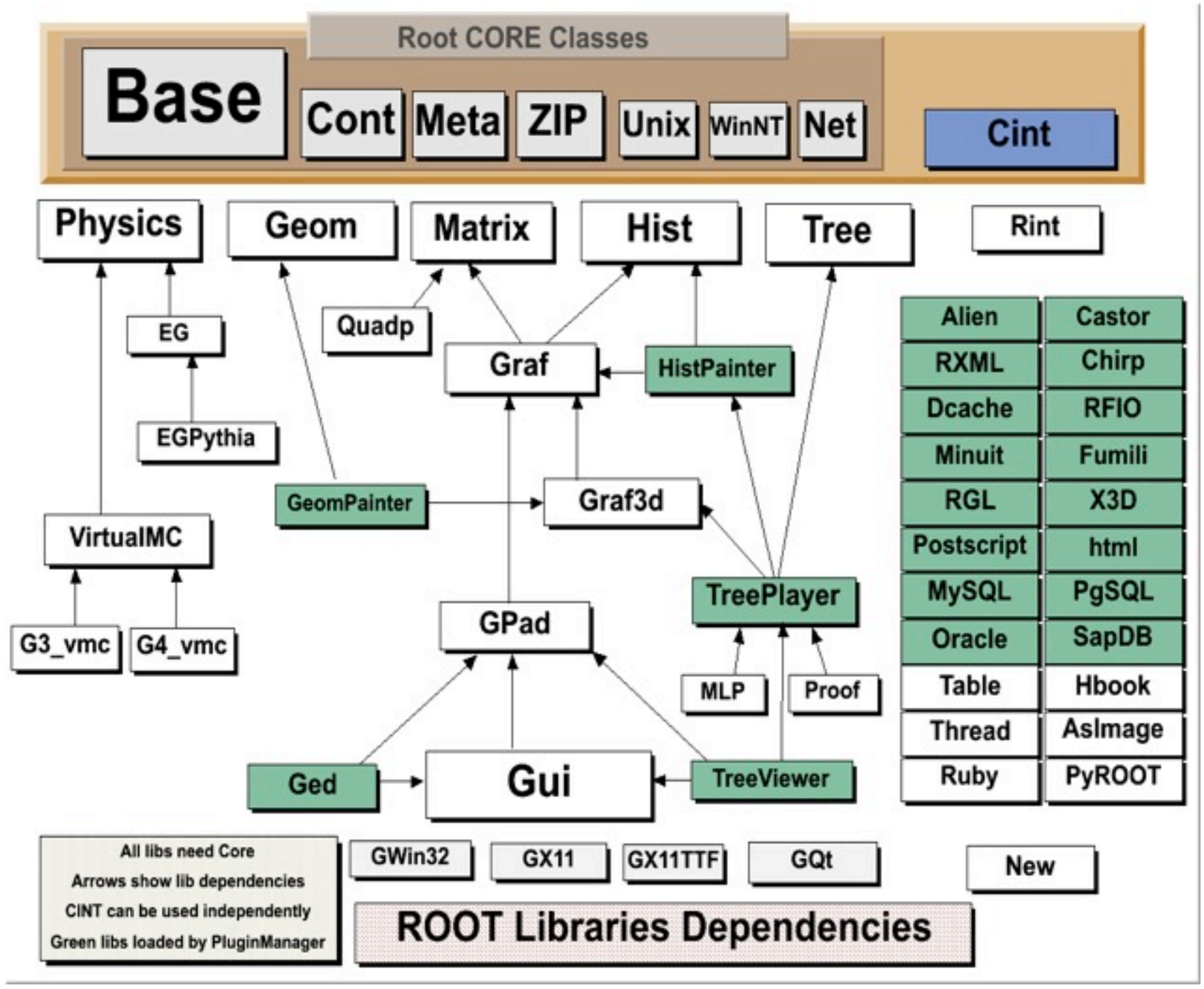


Data Storage: Local, Network

ROOT Libraries

- Overview of ROOT libraries and their dependencies

  - 1,700,000 lines of code.
  - More than 100 shared libraries
  - Fully cross-platform: Unix/ Linux, MacOS and Windows.
  - More than 10000 downloads every month



The Interpreter: CINT

- ROOT is shipped with an interpreter, CINT*
  - C++ not trivial to interpret and not foreseen in the language standard!
- Provides interactive shell.
- Can interpret “macros” (not compiled programs)
  - Rapid prototyping possible.
- ROOT provides also Python bindings:
  - Can use Python interpreter directly after a simple import ROOT !!

```
$ root -b
root [0] 3 * 3
(const int)9
```

New Interpreter: Cling

- Starting from ROOT 6 we will have a new interpreter
  - Cling based on LLVM/Clang
- An interpreter based on a real C++ compiler
  - use the JIT capabilities of Clang to interpret the code

- Can handle templated code at interpreter level
- Can parse also new C++ 11 syntax
  - this would have been impossible with Cling

- A real compiler
  - meaningful error reporting
  - some shortcut of CINT will not be allowed (e.g. using “.” instead of “->”)

Persistency (I/O)

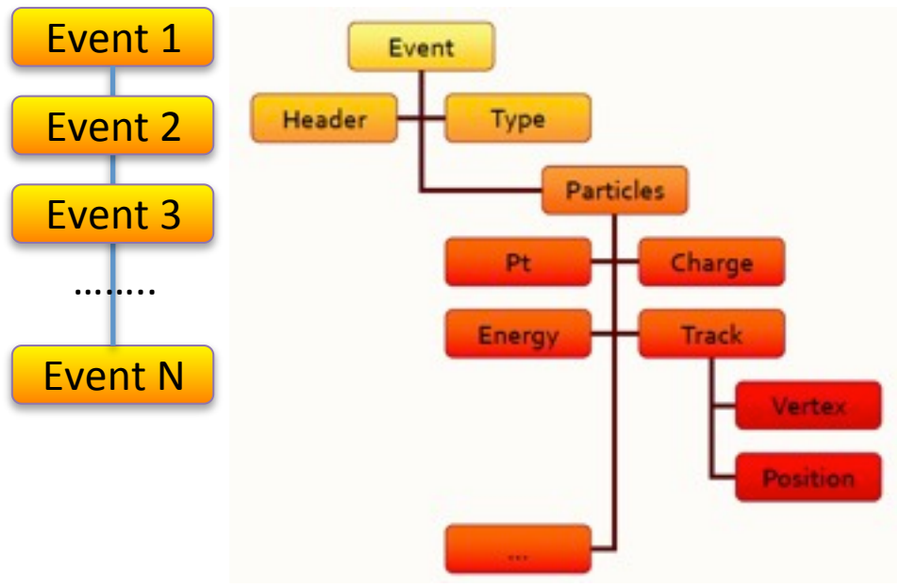
- ROOT offers the possibility to write C++ objects to file
  - Extraordinary: impossible in native C++!
  - Used for petabytes/year rates of LHC detectors.
  - Achieved with serialization of the object using the reflection provided by the CINT dictionary.

Cornerstone of the storage of experimental data

- Single objects, collections, object trees
  - Basically one method for all ROOT objects: `TObject::Write`

x	y	z
-1.10228	-1.79939	4.452622
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

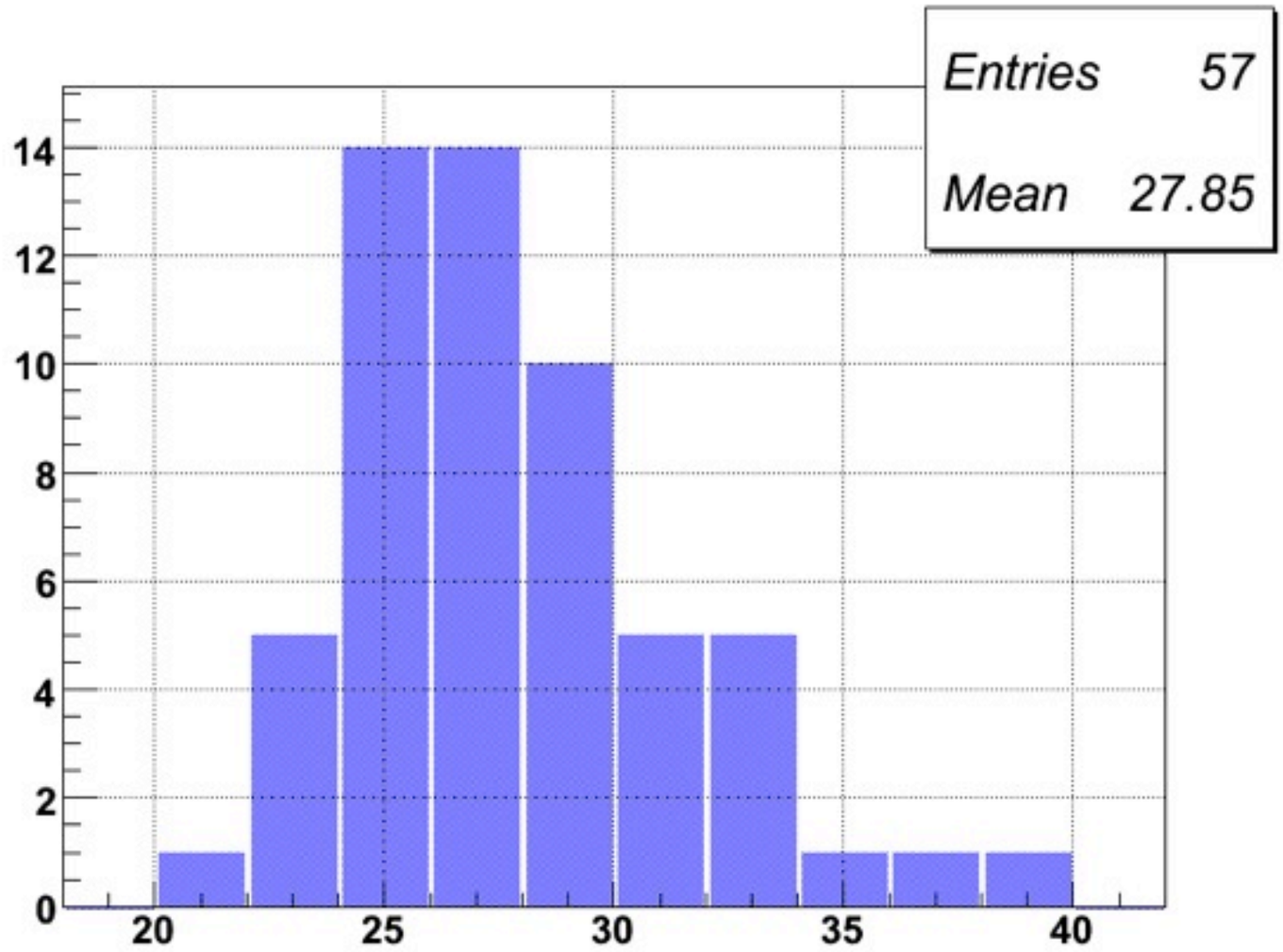
LEP style flat n-tuples evolved in more efficient trees (fast access, read ahead)



Histograms

Table of Ages (binned)

Age	Number
20-22	1
22-24	5
24-26	14
26-28	14
28-30	10
30-32	5
32-34	5
34-36	1
36-38	1
38-40	1



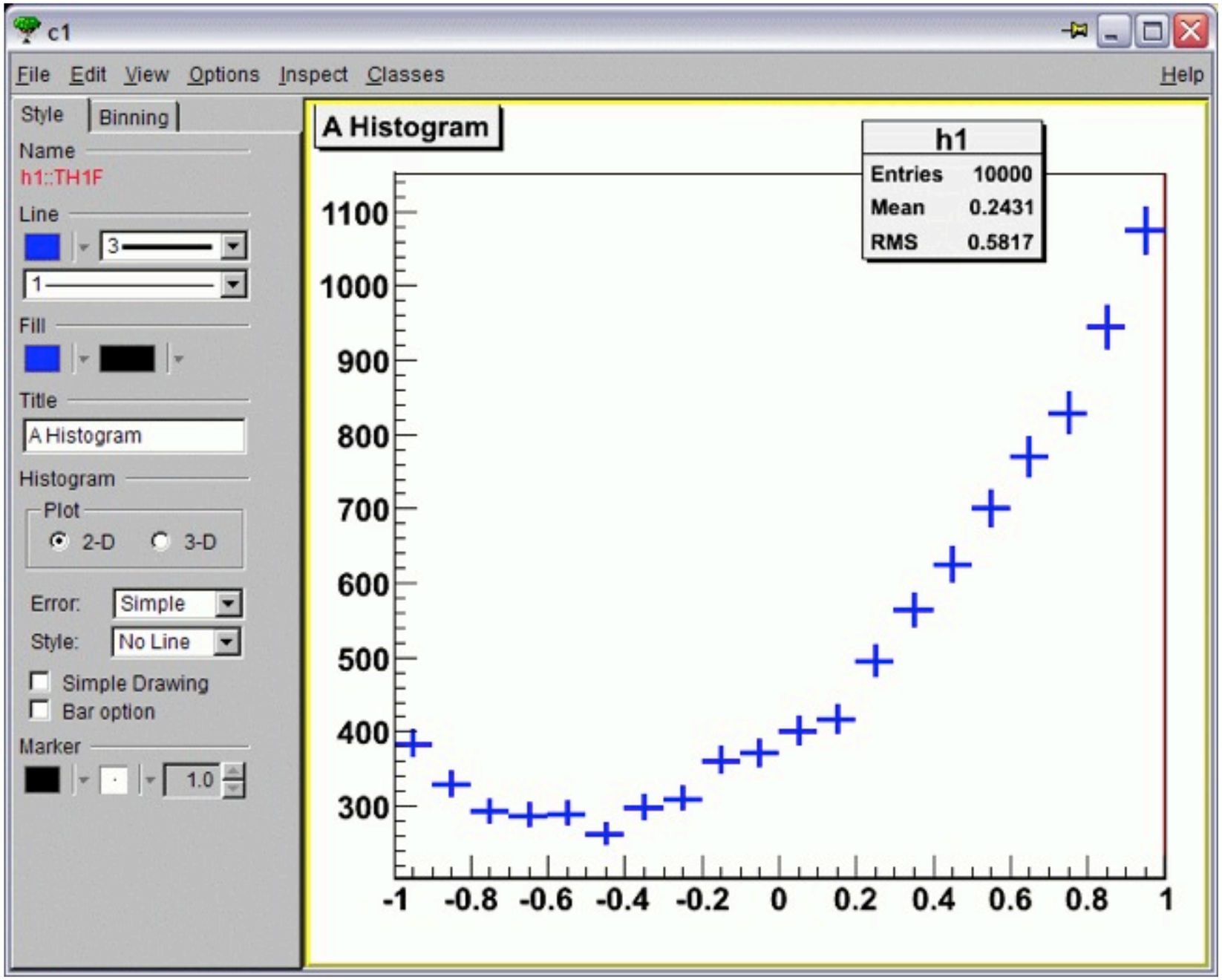
Shows distribution of ages, total number of entries (57 participants) and average: 27 years 10 months 6 days…

Histograms

Analysis result: often a histogram

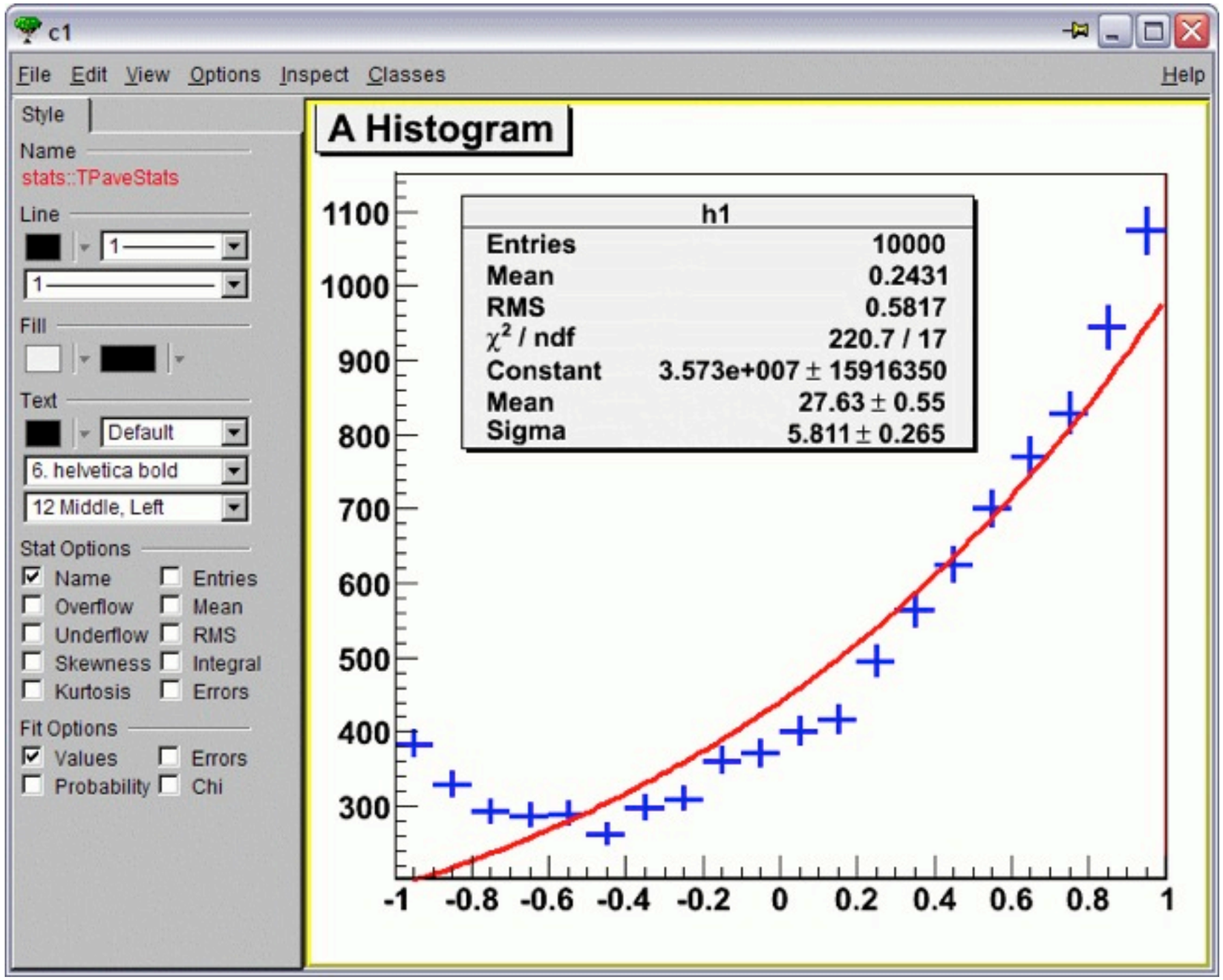
Menu:

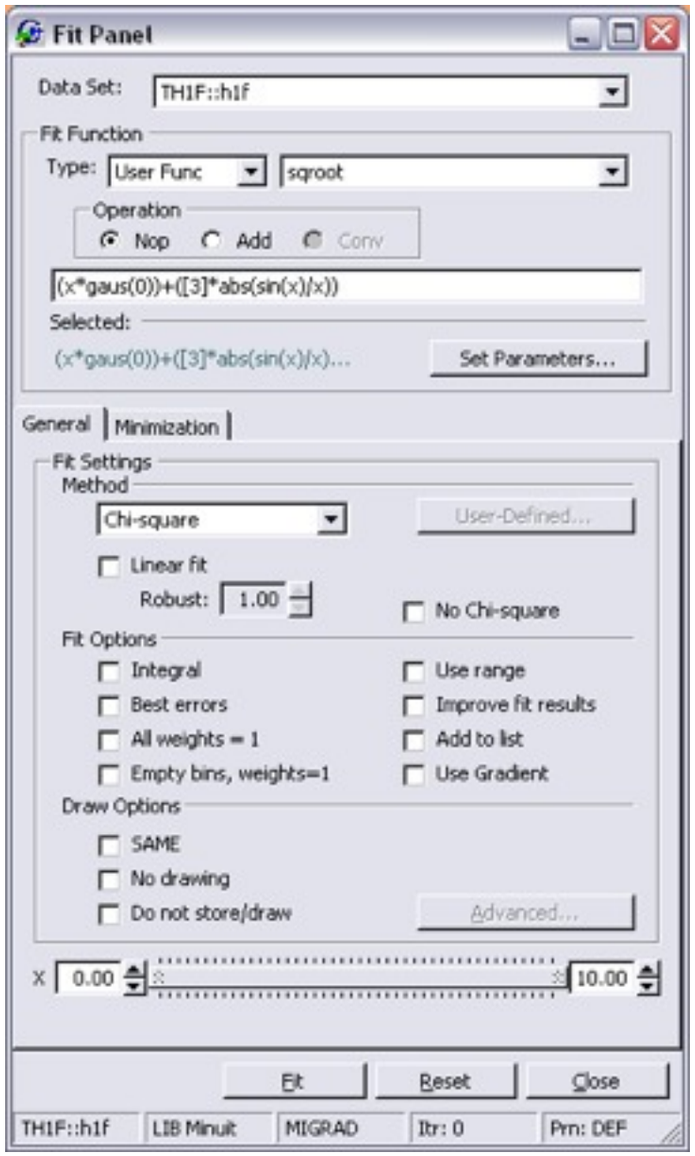
View / Editor



Fitting

Analysis result: often a ***fit*** of a histogram

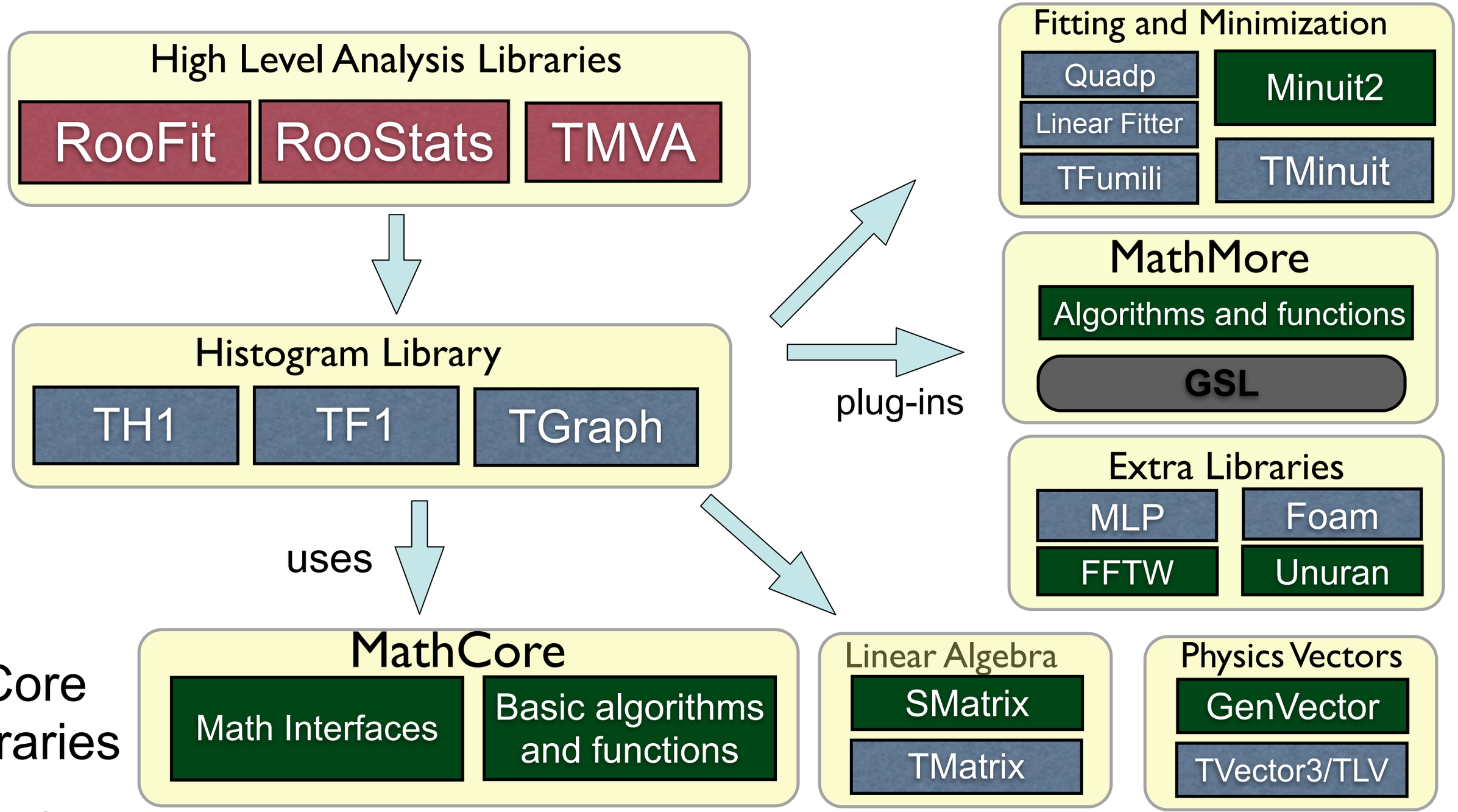




Fit Panel GUI

ROOT Math/Stat Libraries

- ROOT provides a reach set of mathematical libraries and tools needed for event reconstruction, simulation and statistical data analysis



Visualization Techniques in ROOT

The ROOT framework provides many techniques to visualize multi-variable data sets from 2 until N variables.

- 2 variables visualization techniques are used to display Trees, Ntuples, 1D histograms, functions y=f(x), graphs .

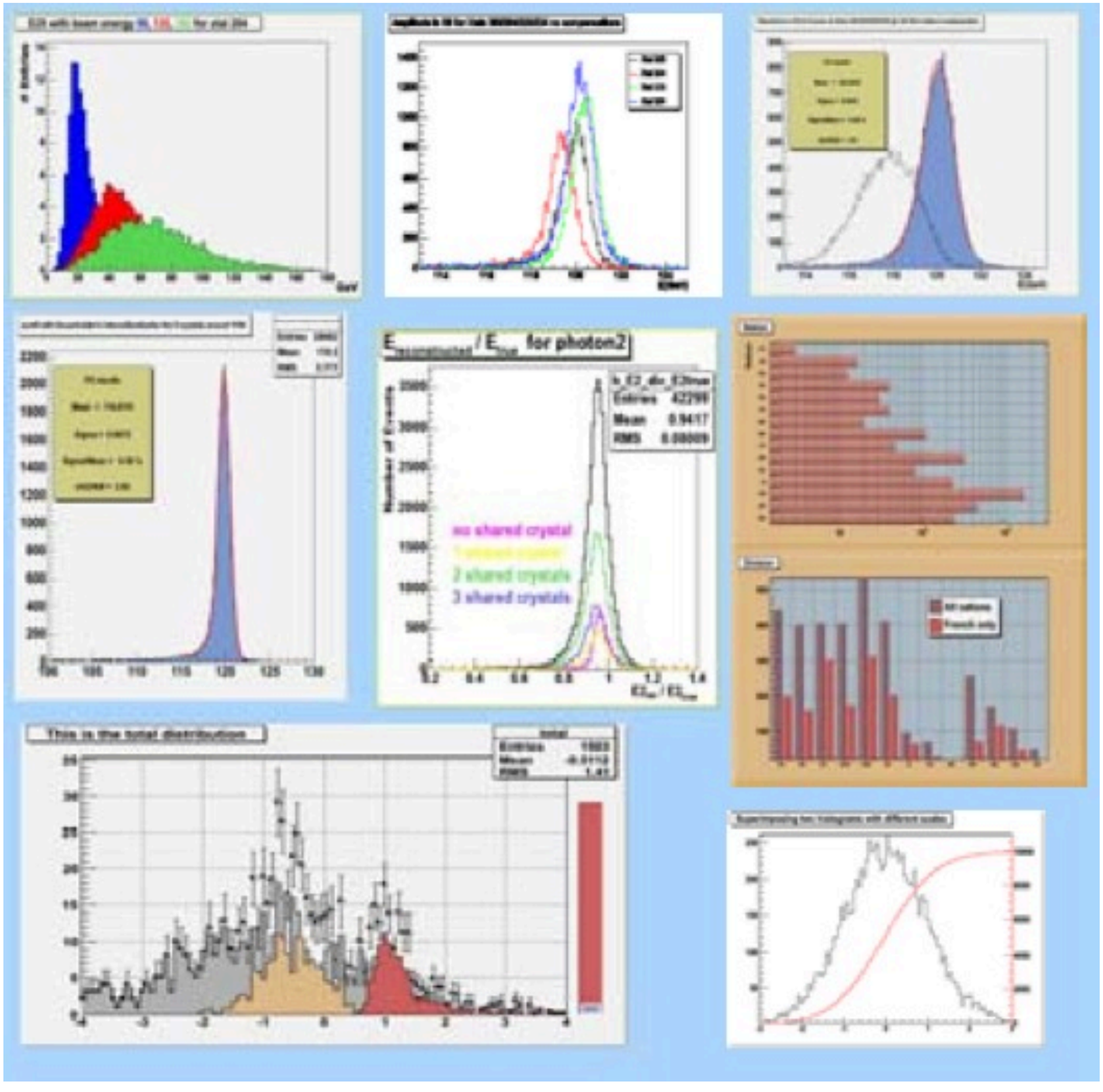
- 3 variables visualization techniques are used to display Trees, Ntuples, 2D histograms, 2D Graphs, 2D functions ...

- 4 variables visualization techniques are used to display Trees, Ntuples, 3D histograms, 3D functions ...

- N variables visualization techniques are used to display Trees and Ntuples ...

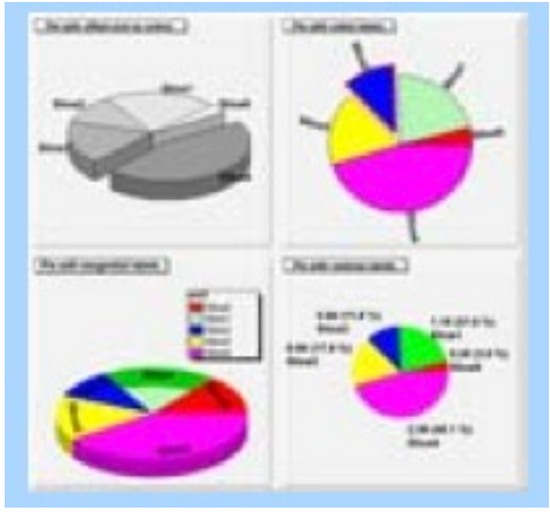
Can save graphics in many formats: `ps,pdf,svg,jpeg,png,c,root`

2 Variables Techniques: Histograms



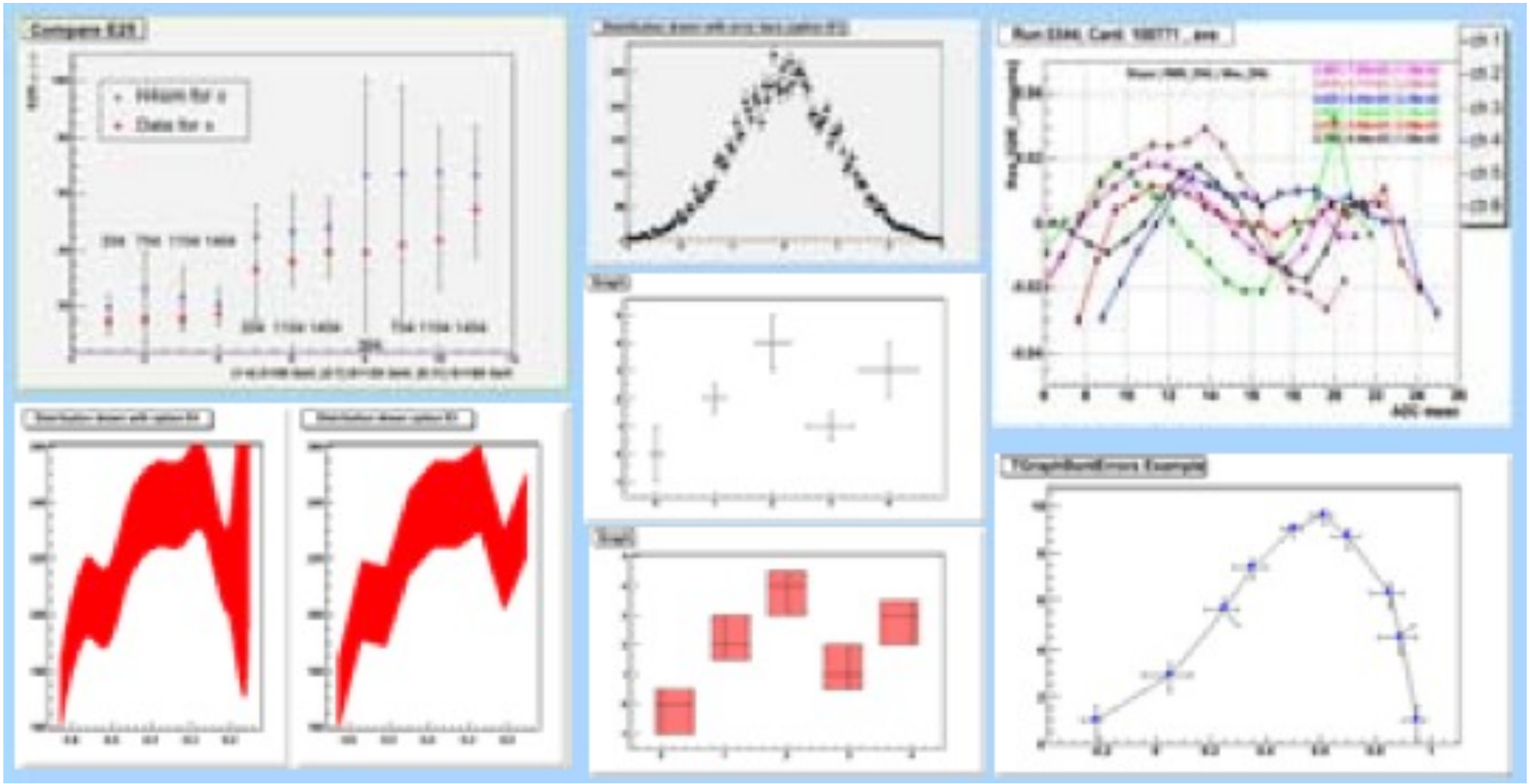
Bar charts and lines are a common way to represent 1D histograms.

Pie charts can be also used to visualize 1D histograms.



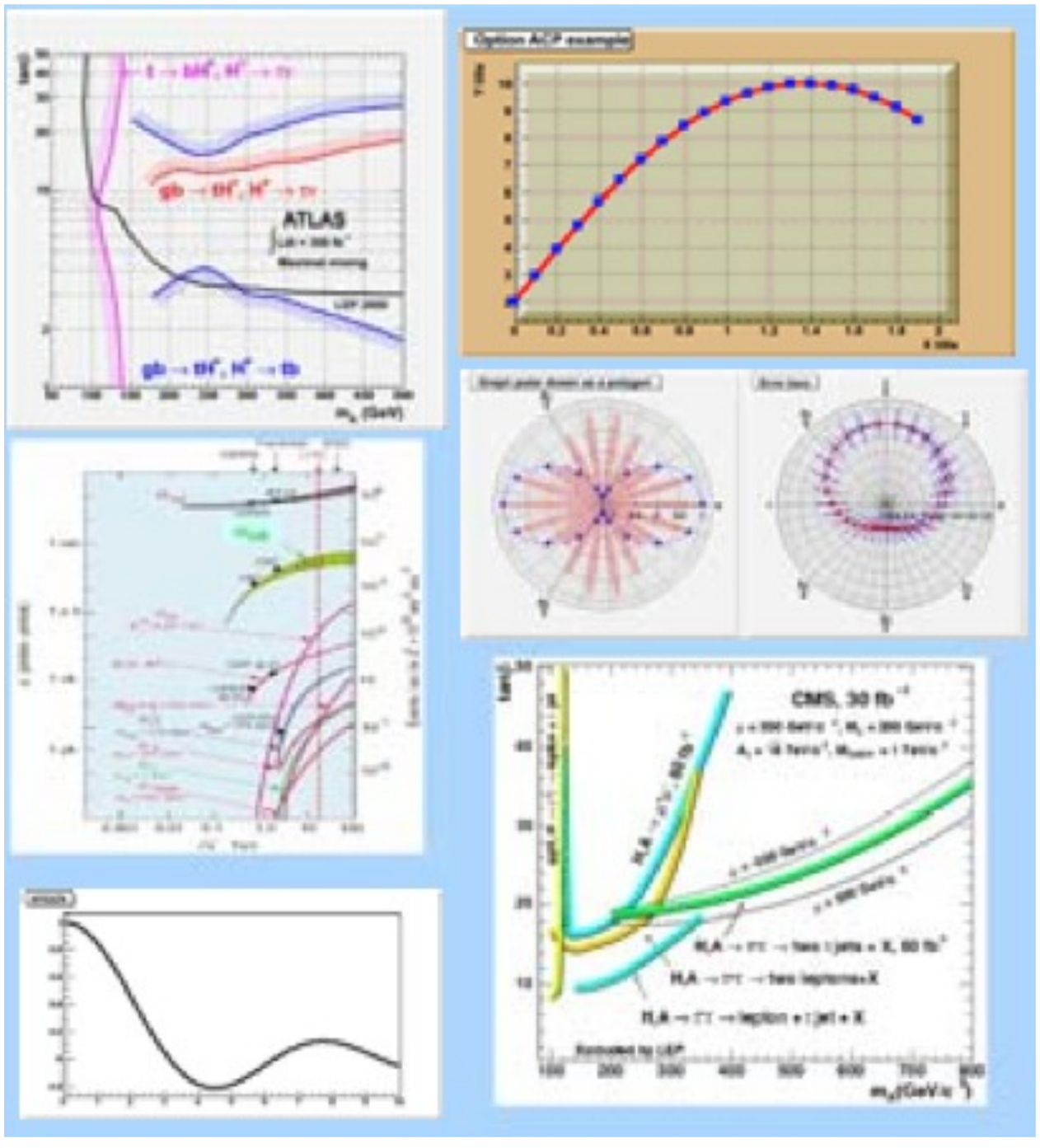
2 Variables Techniques: Errors

Visualization of Variables with Errors (1D Histograms and Graphs)



Errors can be represented as bars, band, rectangles.
They can be symmetric, asymmetric or bent.

2 Variables Techniques: Graphs



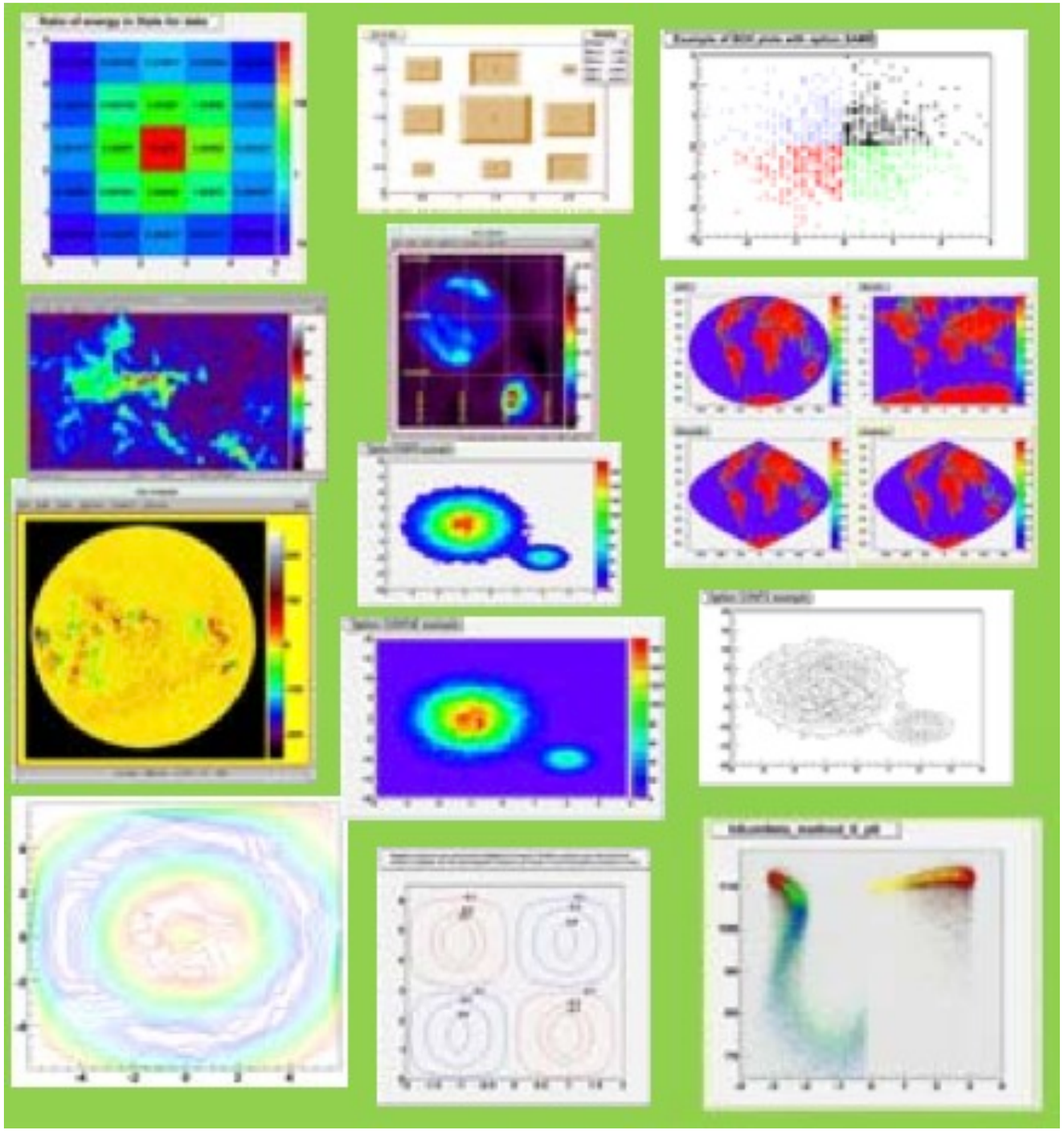
Graphs can be drawn as simple lines, like functions.
Can also visualize exclusion zones.
Can be plotted in polar coordinates.

3 Variables Techniques in 2D

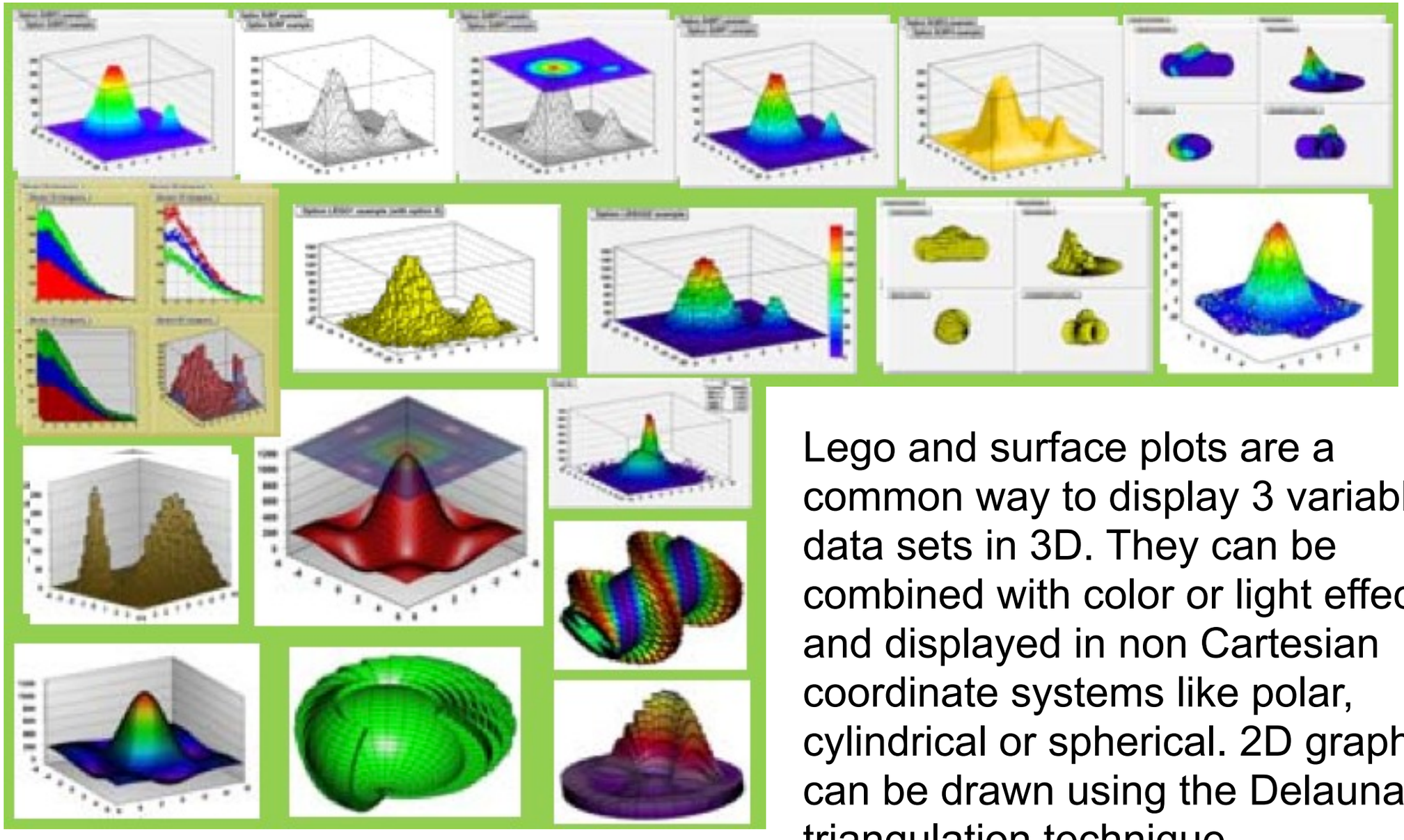
Several techniques to visualize 3 variables data sets (TH2, TGraph2D) in 2 dimensions.

Two variables are mapped on the X and Y axis and the 3rd one on some graphical attributes:

- the color or the size of a box;
- a density of points (scatter plot);
- writing the value of the bin content.
- Using contour plots. Some special projections (like Aitoff) are available to display such contours.



3 Variables Techniques in 3D

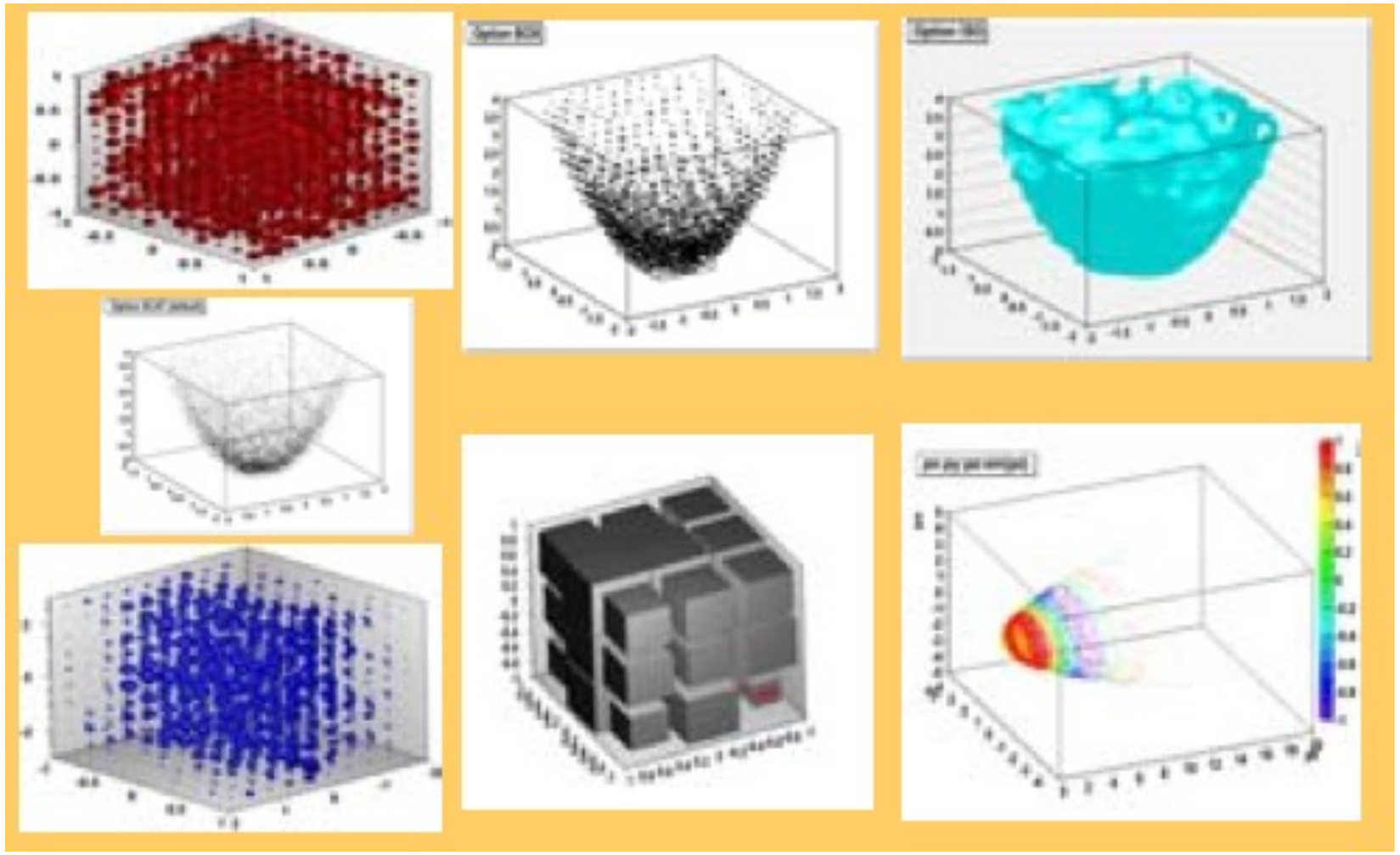


Lego and surface plots are a common way to display 3 variables data sets in 3D. They can be combined with color or light effects and displayed in non Cartesian coordinate systems like polar, cylindrical or spherical. 2D graphs can be drawn using the Delaunay triangulation technique.

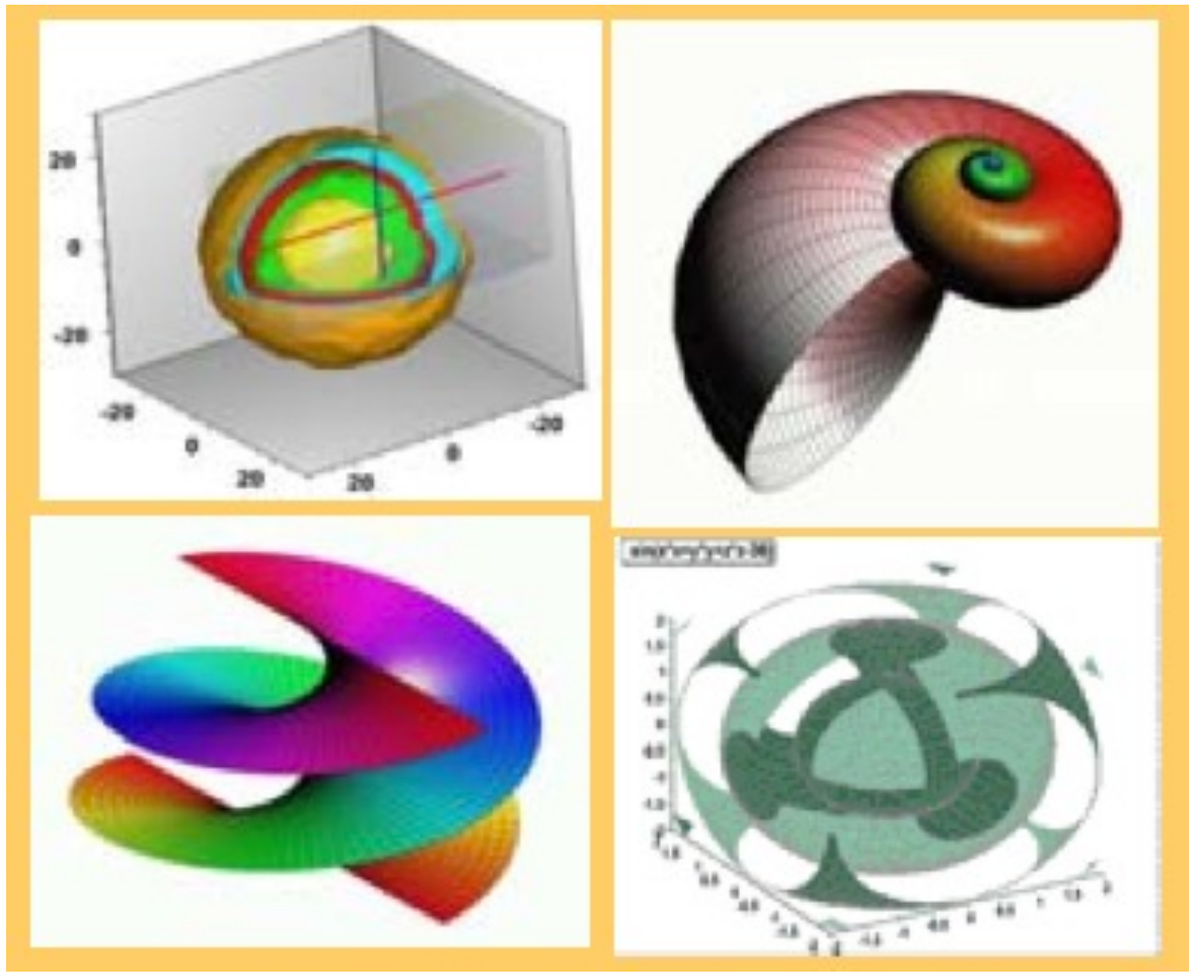
4 Variables Techniques

4 variables data set representations (e.g. TH3, TF3)

Rectangles become boxes or spheres, contour plots become iso-surfaces. The scatter plots (density plots) are drawn in boxes instead of rectangles. The 4th variable can also be mapped on colors. The use of OpenGL allows to enhance the plots' quality and the interactivity.



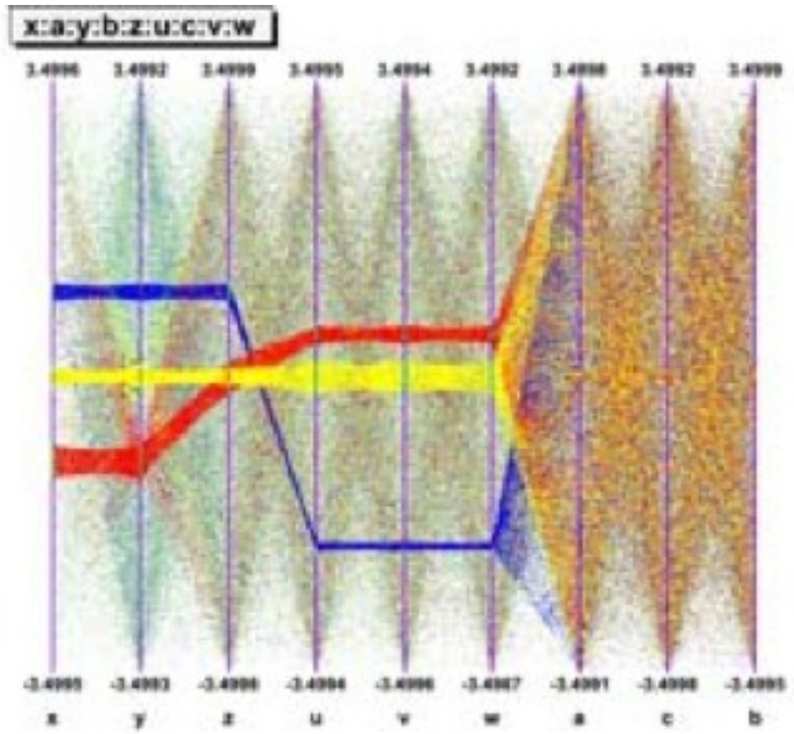
4 Variables Techniques using OpenGL



Functions like t= f(x,y,z) and 3D histograms are 4 variables objects.

ROOT can render using OpenGL. It allows to enhance the plots' quality and the interactivity. Cutting planes, projection and zoom allow to better understand the data set or function.

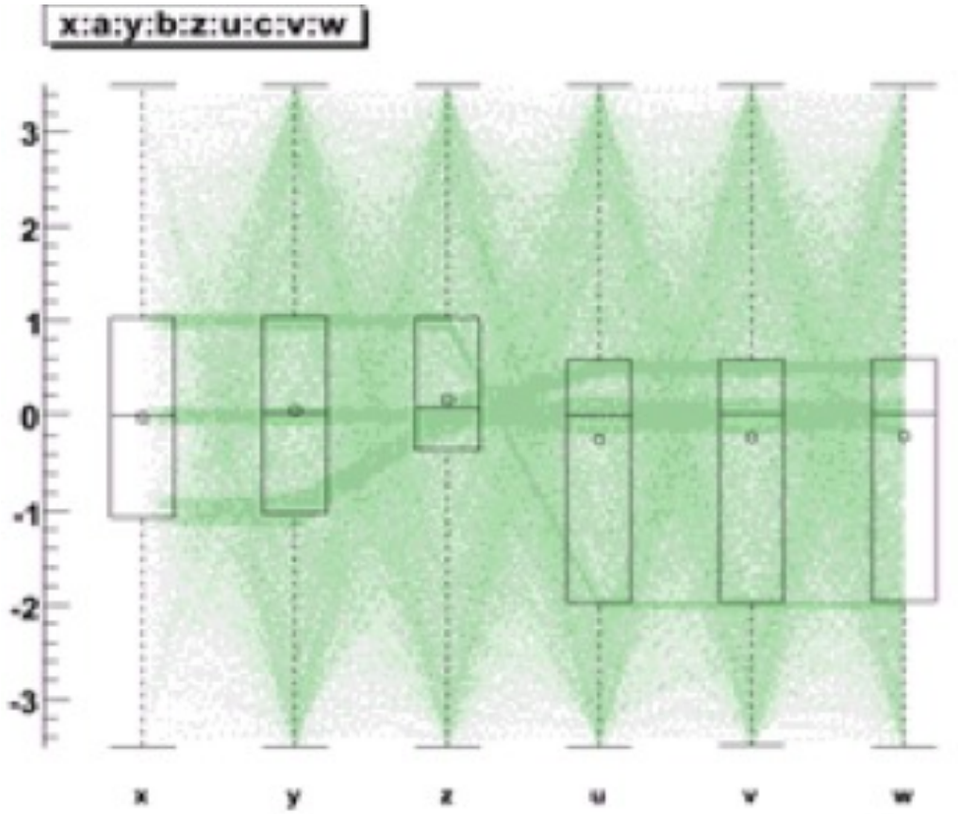
N Variables Techniques



Specific visualization techniques are required for N > 4. ROOT provides:

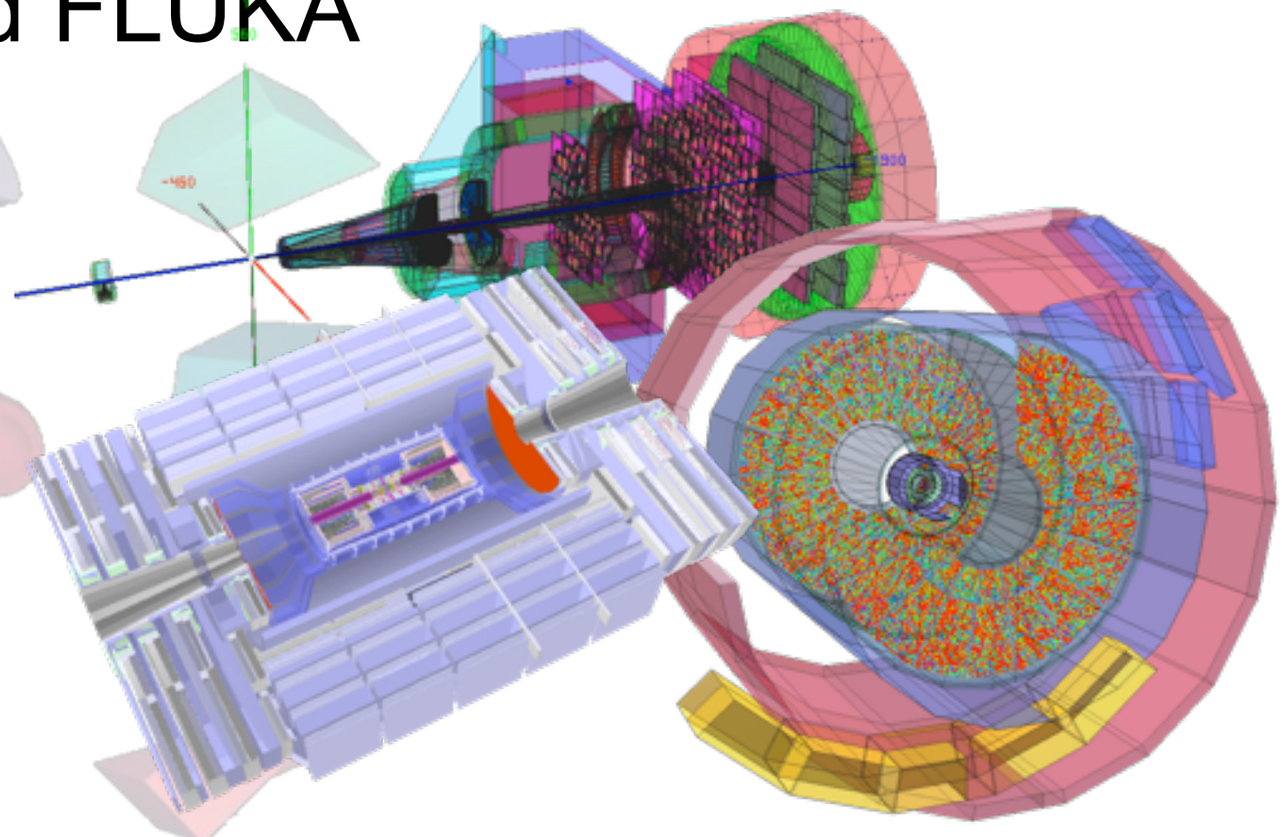
- The parallel coordinates (top)
- the candle plots (right) which can be combined with the parallel coordinates.
- The spider plot (top right).

These three techniques, and in particular the parallel coordinates, require a high level of interactivity to be fully efficient.



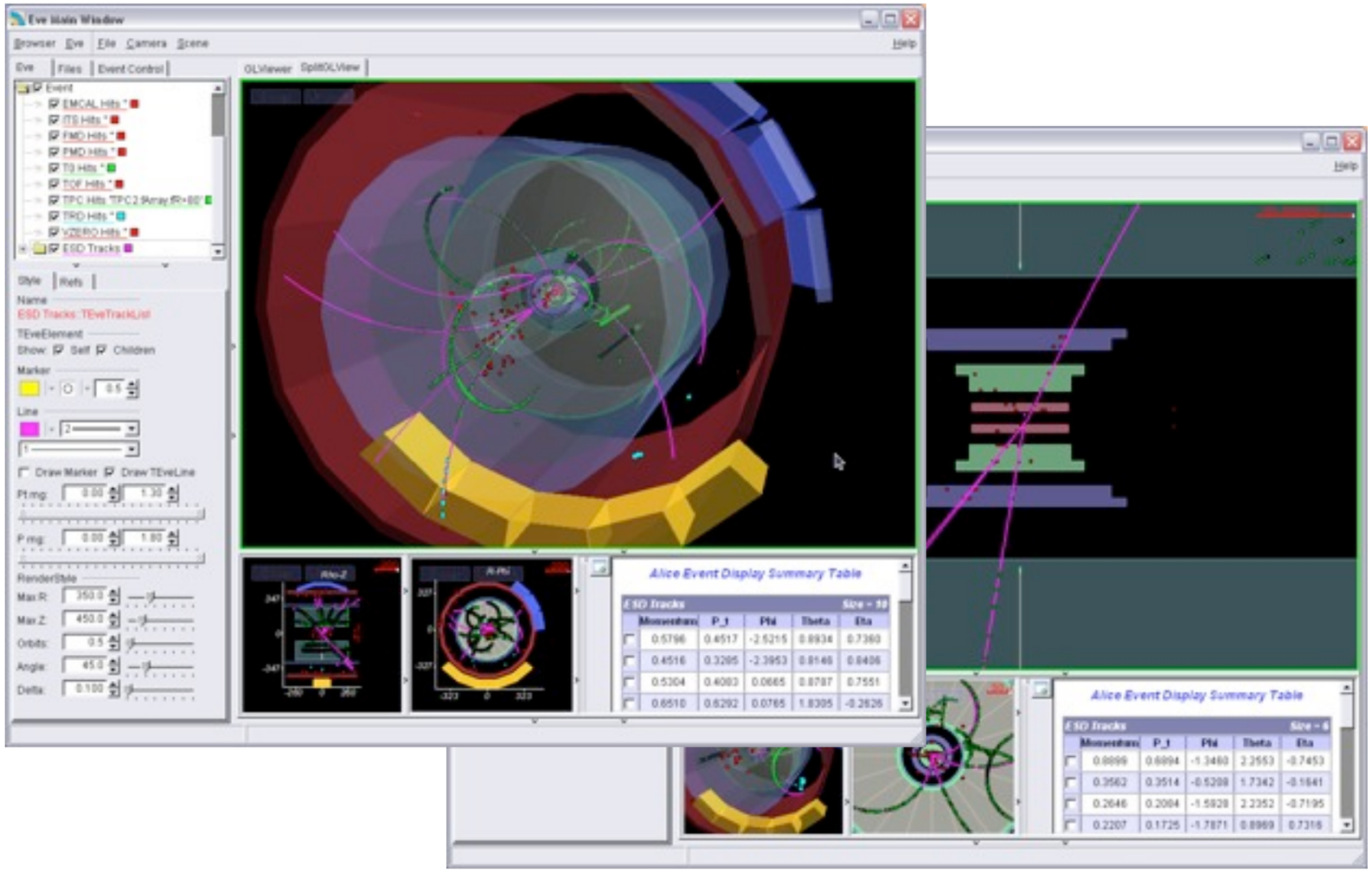
Geometry

- Describes complex detector geometries
- Allows visualization of these detector geometries with e.g. OpenGL
- Optimized particle transport in complex geometries
- Working in correlation with simulation packages such as GEANT3, GEANT4 and FLUKA



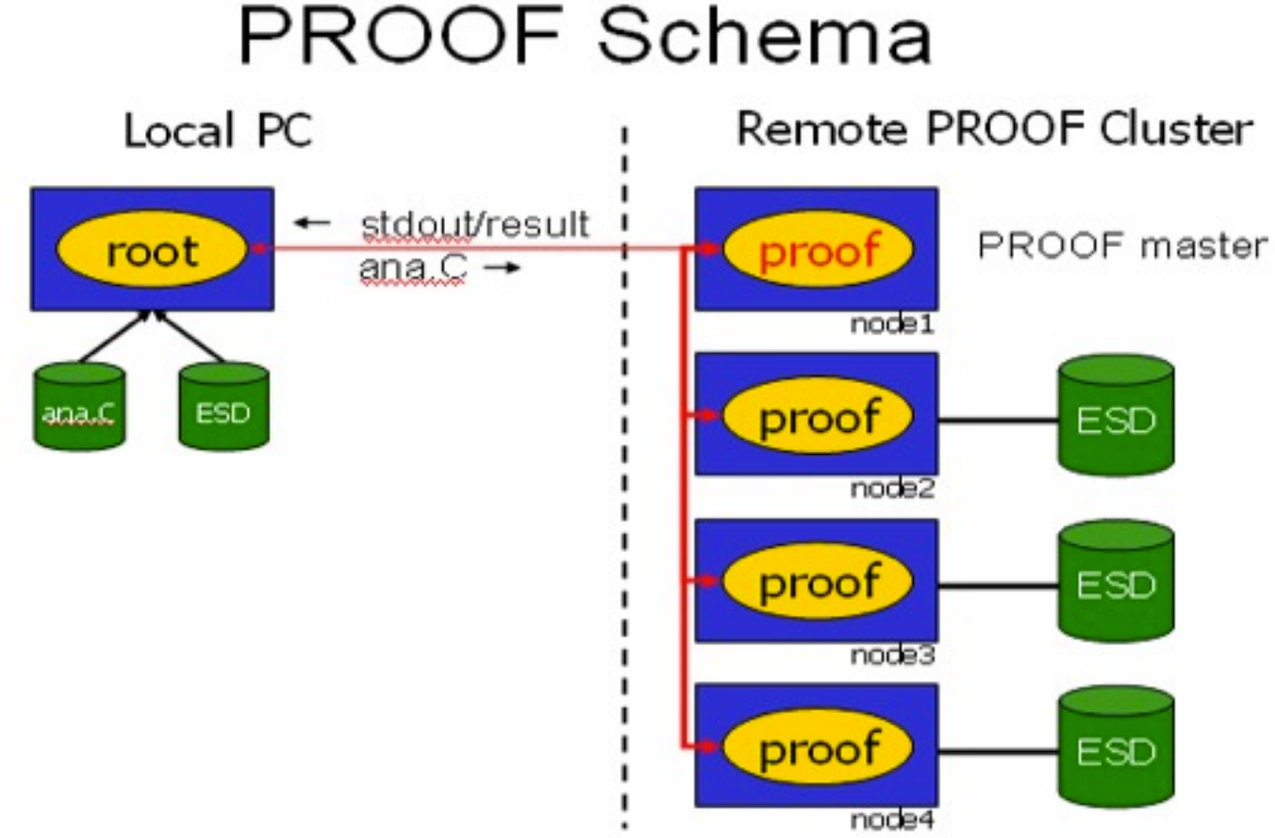
EVE (Event Visualization

- Visualize detector events (tracks, hits,... i.e. physics objects) together with detector geometry (OpenGL)
- Visually interact with the data, e.g. select a particular track and retrieve its physical properties



PROOF

- PROOF: Parallel ROOT Facility
  - Multi-process approach to parallelism
  - A system to run ROOT queries in parallel on a large number of distributed computers
  - Proof-lite: does not need a farm,
    uses all the cores on a desktop machine



Summary

- ROOT: complete analysis framework offering a C++ interpreter, a powerful persistency mechanism, advanced mathematical tools and even a parallel facility
  - it is designed for data analysis of very large shared data sets
  - it is the result of 18 years of cooperation between the development team and thousands of heterogeneous users
  - it is used extensively and profitably by physicists
    - all Higgs discovery plots made with ROOT

- We will now explore in more detail some part of ROOT

Let's Start Using ROOT

Assuming ROOT is now properly installed in your system

Outline

- Starting to work from the ROOT prompt
  - running macro's using CINT and AClic
- Simple ROOT I/O
- Creating and Reading a ROOT Tree object
- Data Analysis with ROOT Trees

We will have interleaving lectures and exercises

Documentation is all available at the ROOT Web site

http://root.cern.ch

ROOT Download & Installation

- Download from ROOT Web site http://root.cern.ch
- Binaries for Linux, MacOS and Windows
- Source files can be built with:
  - `./configure`
    `make`
  - CMake
  - see the instructions on the Web site for building from sources

- *For the tutorials ROOT should have been already installed !*

Starting Up ROOT

ROOT is prompt-based. Launch ROOT:

```
$ root
```

The ROOT Prompt will appear:

```
root [0] _
```

It “speaks” C++ (using CINT interpreter):

```
root [0] sqrt(42)
(const double)6.48074069840786038e+00
root [1] sin(val)
(const double)1.69182349066996029e-01
root [2] TF1 * f1 = new TF1(“f1”,”sin(x)/x”,0,10);
root [3] f1->Draw();
```

Running Code

Macro: a file that is interpreted by CINT

```
int mymacro(int value)
{
  int ret = 42;
  ret += value;
  return ret;
}
```

- Create a new file, `mymacro.C`
- Edit the file and include these above lines.
- Execute from the root prompt:

```
root [0] .x mymacro.C(42)
```

Compiled versus Interpeter

- Why compile the macro ?
  - Faster execution, CINT has limitations, validate code.
  - With Cling compilation will be done on the flight
    - Note: CINT allows for some non-C++ syntax
      (e.g. using "->" instead of ".")
      This errors will not be allowed anymore in Cling

- Why interpret ?
  - Faster editing (rapid prototyping)
    - e.g. no need to add all required include files in macro

- ACLIC compilation
  - platform independent
  - no need anymore to use Makefile's

Controlling ROOT

- Useful commands from the ROOT prompt:
  - quit ROOT

```
root [1] .q
```

  - to get the list of available commands

```
root [1] .?
```

  - to access the shell of the OS (e.g UNIX or MS/DOS)

```
root [1] .!  <OS_command>
```

e.g.: .! pwd

  - to execute a macro (add a + at the end for compiling with ACLIC)

```
root [1] .x <file_name>
```

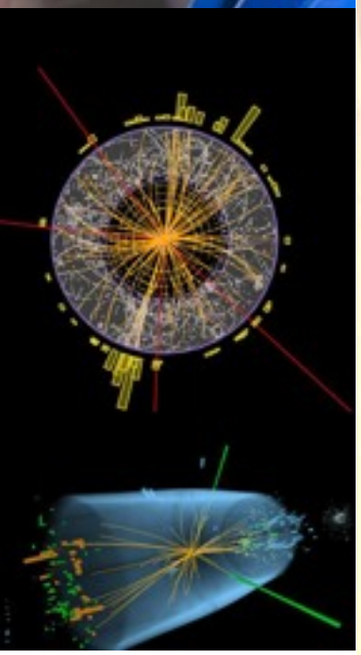
e.g.: .x mymacro.C
.x mymacro.C+

  - to load a macro

```
root [1] .L <file_name>
```

e.g.: .L mymacro.C
.L mymacro.C+

Introduction to ROOT I/O



Dr Lorenzo Moneta
CERN PH-SFT
CH-1211 Geneva 23
sftweb.cern.ch
root.cern.ch

Saving ROOT Objects
Streaming, Reflection, TFile, Schema Evolution

Saving Objects

Cannot do in C++:

```
TNamed* o = new TNamed("name","title");
std::write("file.bin", "obj1", o);
TNamed* 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: 15 PetaBytes / year of processed data (i.e. data that will be read)

Saving Objects – Saving Types

What's needed?

```
TNamed* o = new TNamed("name","title");
std::write("file.bin", "obj1", o);
TNamed* p = std::read("file.bin", "obj1");
p->GetName();
```

Cannot do in C++

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

Store *data members* of TNamed: serialization

1) type of object: runtime-type-information RTTI
2) data members for the type: reflection
3) where data members are in memory: introspection
4) read their values from memory, write to disk: raw I/O

Complex task, and C++ is not your friend.

Reflection

Need type description (aka *reflection*)

1. types, sizes, members

`TMyClass` is a class.

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

Members:

- "`fFloat`", type `float`, size 4 bytes
- "`fLong`", type `Long64_t`, size 8 bytes

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!

ROOT Basic Data Types

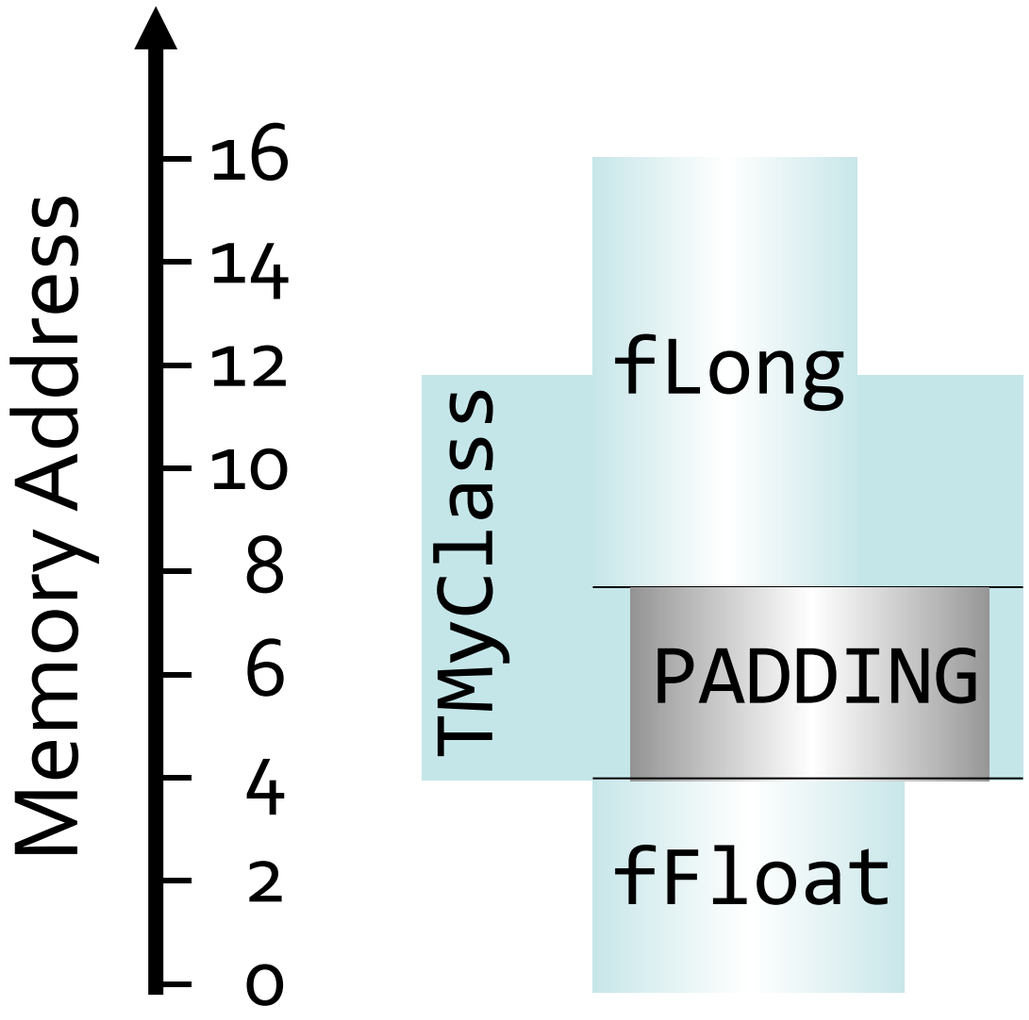
Solution: ROOT typedefs

Signed	Unsigned	sizeof [bytes]
`Char_t`	`UChar_t`	1
`Short_t`	`UShort_t`	2
`Int_t`	`UInt_t`	4
`Long64_t`	`ULong64_t`	8
`Double32_t`		float on disk, double in RAM

Reflection

Need type description (platform and compiler 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

C++ Is Not Java

Lesson: need reflection!

Where from?

Java: get data members with

```
Class.forName("MyClass").getFields()
```

C++: get data members with

– oops. Not part of C++.

ROOT And Reflection

- Simply use ACLiC:

```
.L MyCode.cxx+
```

- Creates library with reflection data ("dictionary") of all types defined in MyCode.cxx!
- For types defined only in external include files, add in code:

```
#ifdef __MAKECINT__
#pragma link C++ class MyClass+;
#endif
```

- Dictionary are needed also for interpreter
- ROOT has dictionary for all its types

LinkDef file and rootcint

- To make a library containing the dictionary of your class
  - create a LinkDef.h file containing:

```
#ifdef __CINT__
#pragma link C++ class MyClass+;
#endif
```

  - call rootcint command:

```
> rootcint -f MyClassDict.cxx -c MyClass.h LinkDef.h
```

  - compile in a library (libMyClass.so) the generated dictionary file (MyClass.cxx) together with the implementation of your class

Saving Objects in ROOT

- Use the `TFile` class
  - we need first to create the class, which opens the file

```
TFile* f = TFile::Open("file.root","NEW");
```

use option "`RECREATE`" if the file already exists

- Write an object deriving from `TObject`:

```
object->Write("optionalName")
```

if the optionalName is not given the object will be written in the file with its original name (`object->GetName()`)

- For any other object (but with dictionary)

```
f->WriteObject(object, "name");
```

TFile Class

- ROOT stores objects in TFiles:

```
TFile* f = TFile::Open("file.root", "NEW");
```

- TFile behaves like file system:

```
f->mkdir("dir");
```

- TFile has a current directory:

```
f->cd("dir");
```

- You can browse the content:

```
f->ls();
 TFile**         file.root
 TFile*          file.root
  TDirectoryFile*      dir   dir
  KEY: TDirectoryFile  dir;1 dir
```

Saving Histogram in a File

- How to save objects in a file

```
TFile* f = TFile::Open("myfile.root",”NEW”);
TH1D* h1 = new TH1D(“h1”,”h1”,100,-5.,5.);

h1->FillRandom(“gaus”);  // fill histogram with random data

h1->Write();

delete f;
```

- TFile compresses data using ZIP

```
h1->Write();
f->GetCompressionFactor()
(Float_t)1.68554687500000000e+00
```

Where is My Histogram ?

- All histograms and trees are owned by `TFile` which acts like a scope
- After closing the file (i.e when the file object is deleted) also the histogram, trees and graphs objects are deleted
- This code will crash ROOT:

```
TFile* f = TFile::Open("myfile.root",”RECREATE”);

TH1D* h1 = new TH1D(“h1”,”h1”,100,-5.,5.);

delete f;

h1->Draw(); // will crash - DO NOT DO IT!!!

 *** Break *** segmentation violation
```

- Other objects (e.g graphs) will be still there and can be accessed afterwards
- This can be changed with `TH1::AddDirectory(false);`

Risks With I/O

Risks With I/O

Physicists can loop a lot:

Risks With I/O

Physicists can loop a lot:

For each particle collision

Risks With I/O

Physicists can loop a lot:

For each particle collision

For each particle created

Risks With I/O

Physicists can loop a lot:

For each particle collision

For each particle created

For each detector module

Risks With I/O

Physicists can loop a lot:

For each particle collision

For each particle created

For each detector module

Do something.

Risks With I/O

Physicists can loop a lot:

For each particle collision

For each particle created

For each detector module

Do something.

Physicists can loose a lot:

Risks With I/O

Physicists can loop a lot:

For each particle collision

For each particle created

For each detector module

Do something.

Physicists can loose a lot:

Run for hours…

Risks With I/O

Physicists can loop a lot:

For each particle collision

For each particle created

For each detector module

Do something.

Physicists can loose a lot:

Run for hours…

Crash.

Risks With I/O

Physicists can loop a lot:

For each particle collision

For each particle created

For each detector module

Do something.

Physicists can loose a lot:

Run for hours…

Crash.

Everything lost.

Name Cycles

Create snapshots regularly:

MyObject;1

Name Cycles

Create snapshots regularly:

MyObject;1

MyObject;2

Name Cycles

Create snapshots regularly:

MyObject;1

MyObject;2

…

Name Cycles

Create snapshots regularly:

MyObject;1

MyObject;2

…

MyObject;5427

Name Cycles

Create snapshots regularly:

MyObject;1

MyObject;2

…

MyObject;5427

MyObject

Name Cycles

Create snapshots regularly:

MyObject;1

MyObject;2

…

MyObject;5427

MyObject

Name Cycles

Create snapshots regularly:

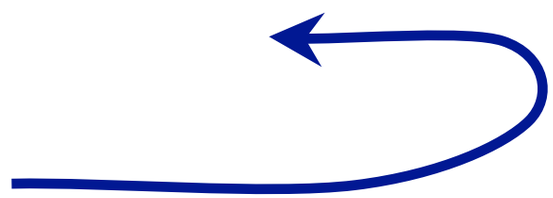
MyObject;1

MyObject;2

…

MyObject;5427

MyObject



Write() does not replace but append!

but see documentation TObject::Write()

Reading a File

- Reading is simple:

```
TFile* f = TFile::Open("myfile.root");
TH1* h = 0;
f->GetObject("h", h);
h->Draw();
delete f;
```

- Can also use
  - `TH1 * h = (TH1*) f->Get("h1");`
  - `TH1 * h = (TH1*) f->GetObjectChecked("h1","TH1");`
    - which returns a null pointer if the read object is not of the right type
- Remember:
  - TFile owns the histogram
  - the histogram is gone when the file is closed
  - to change this add `TH1::AddDirectory(false)` in `root_logon.C`

Ownership And TFiles

Separate TFile and histograms:

```
TFile* f = TFile::Open("myfile.root");
TH1F* h = 0;
TH1::AddDirectory(kFALSE);
f->GetObject("h", h);
h->Draw();
delete f;
```

… and h will stay around.

Put in root_logon.C in current directory to be executed when root starts

Changing Class – The Problem

Things change:

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

Changing Class – The Problem

Things change:

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

Inconsistent reflection data, mismatch in memory, on disk

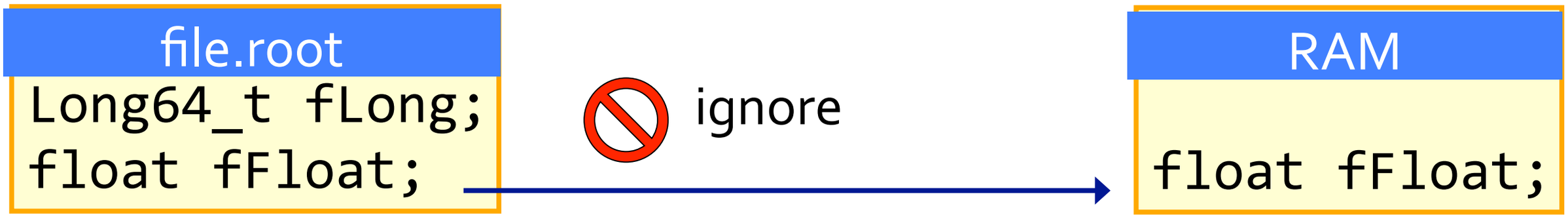
Objects written with old version cannot be read

Need to store reflection with data to detect!

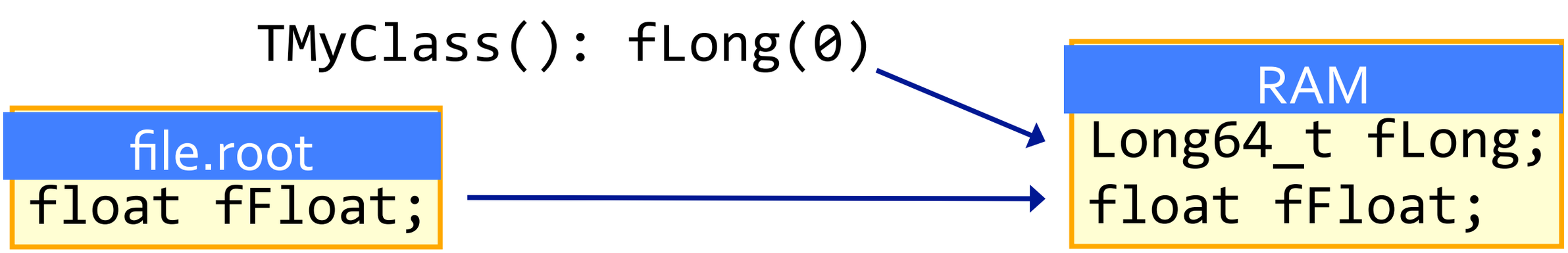
Schema Evolution

Simple rules to convert disk to memory layout

1. skip removed members



2. default-initialize added members



3. convert members where possible
   - a conversion function could be supplied

Class Version

ClassDef() macro

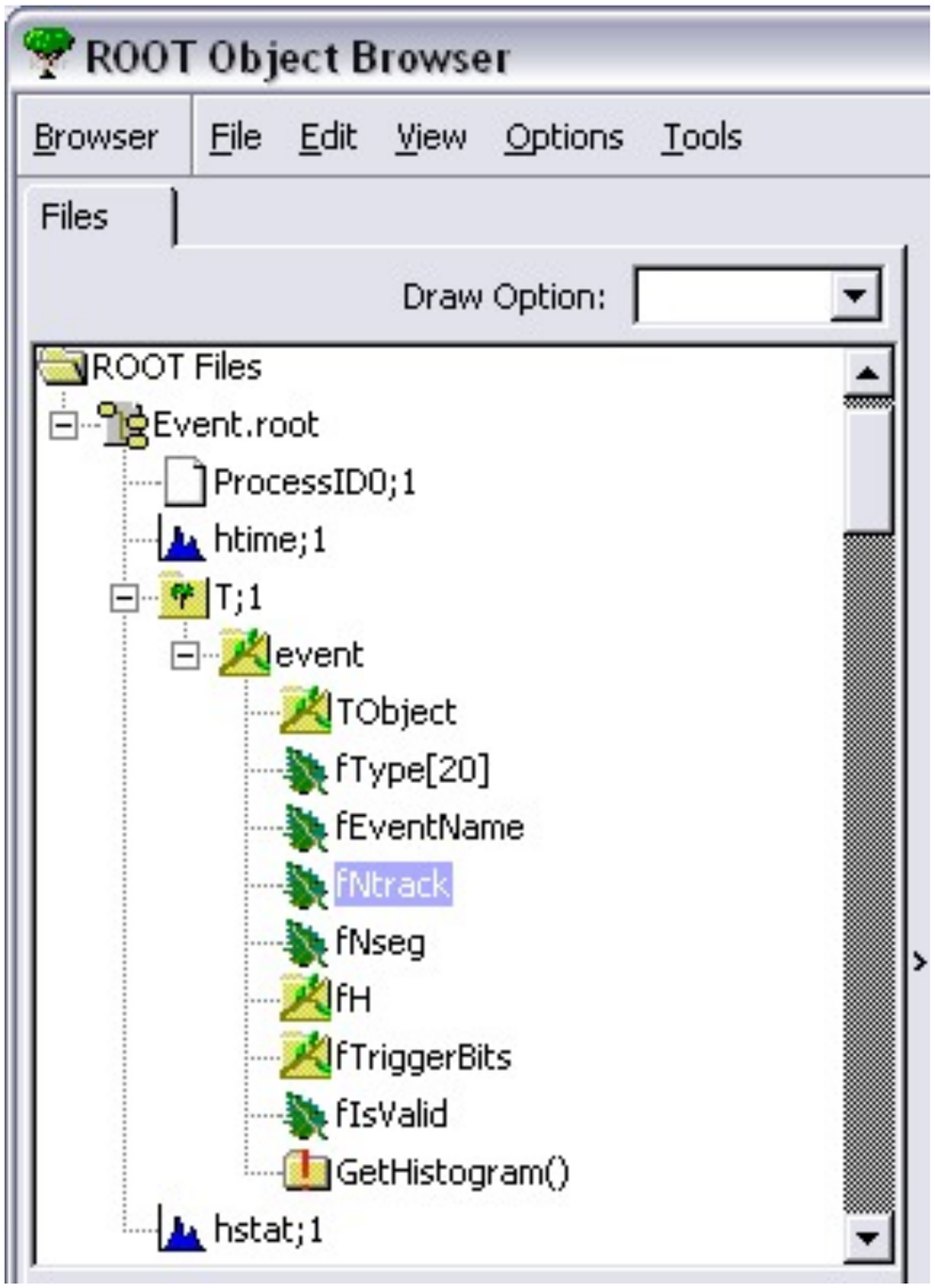
Use version number to identify layout:

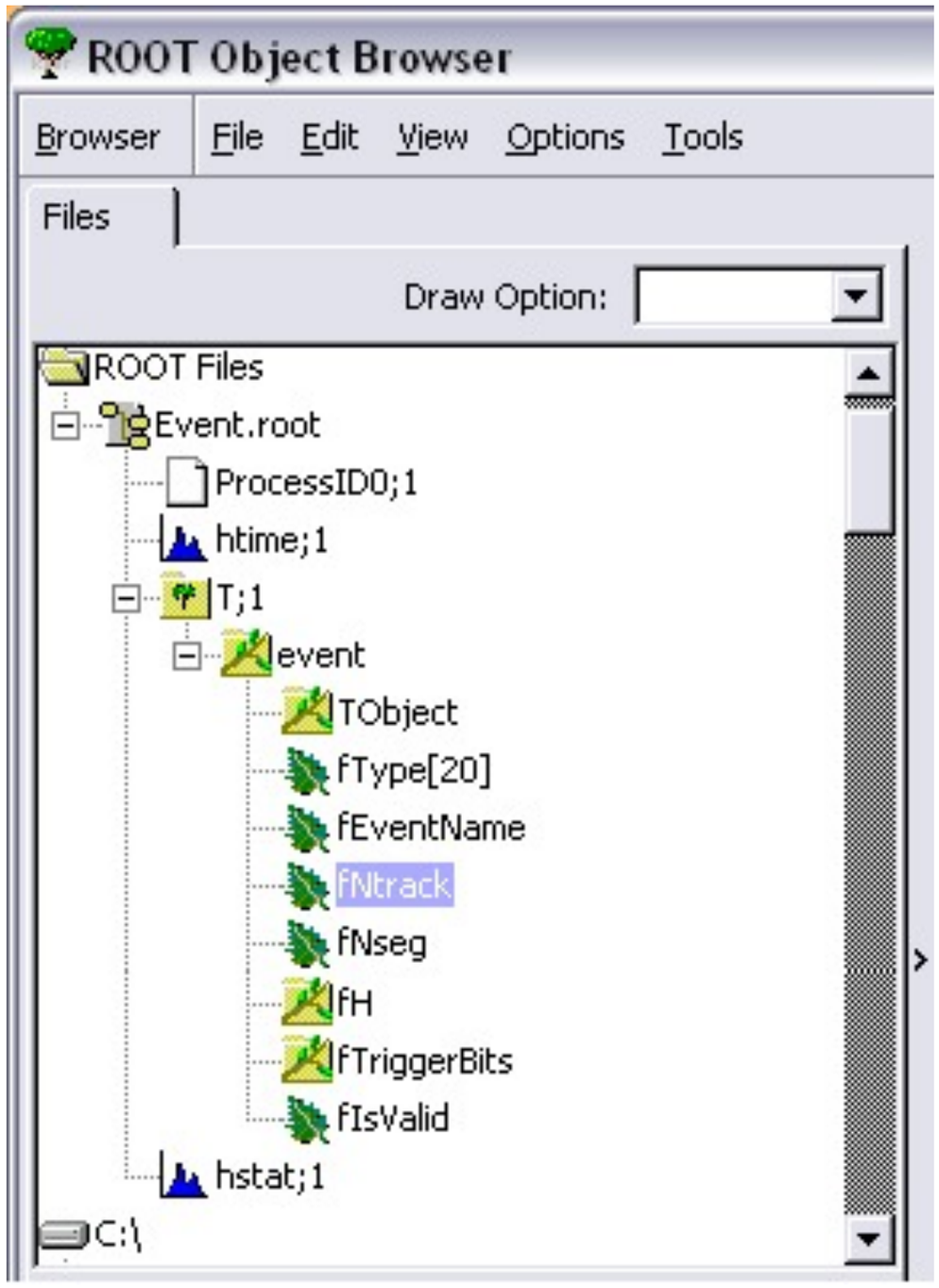
```
class TMyClass: public TObject {
public:
  TMyClass(): fLong(0), fFloat(0.) {}
  virtual ~TMyClass() {}
  ...
  ClassDef(TMyClass,1); // example class
};
```

Can have multiple class versions in same file

Reading Files

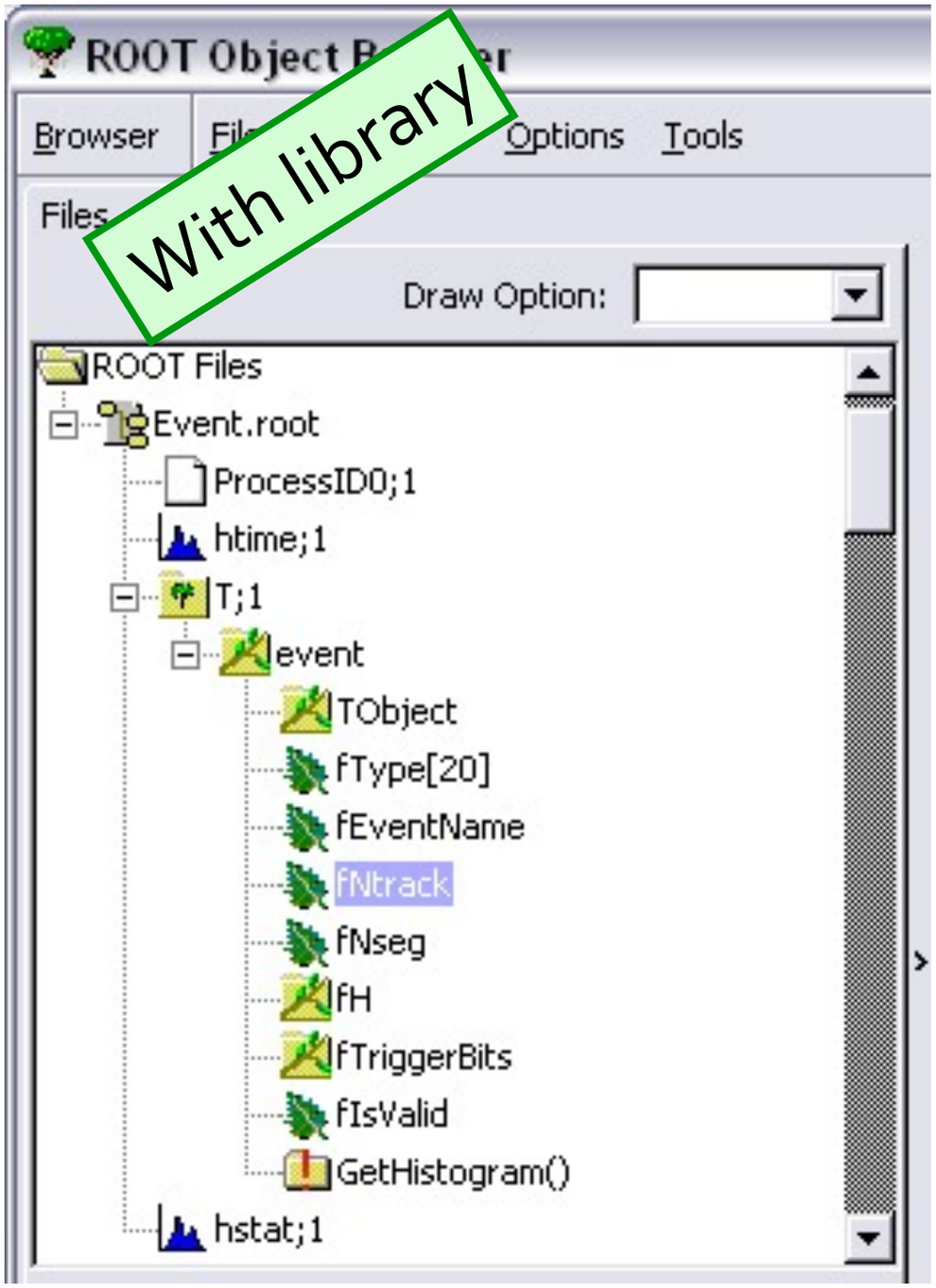
Files store reflection and data: need no library!

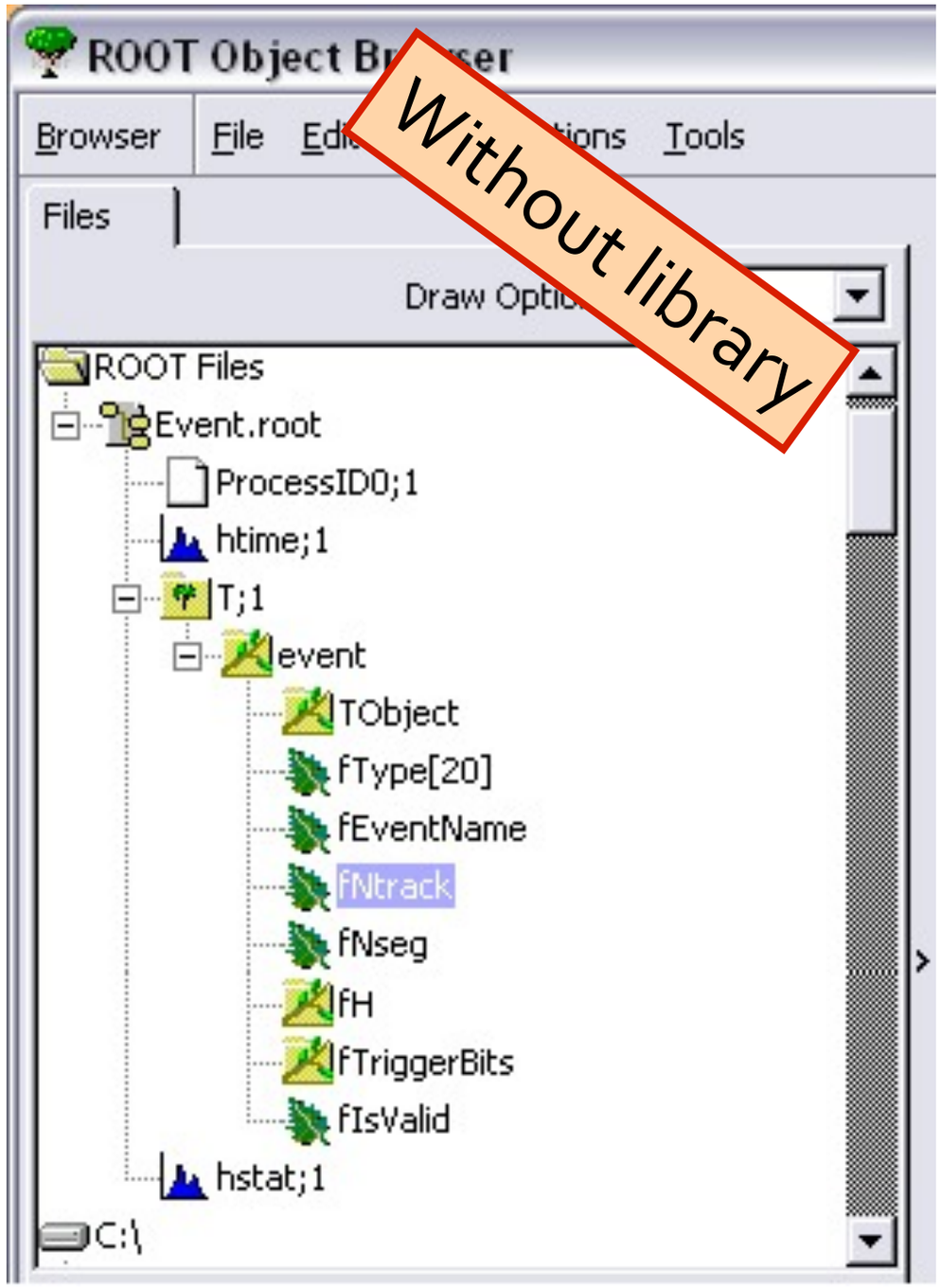




Reading Files

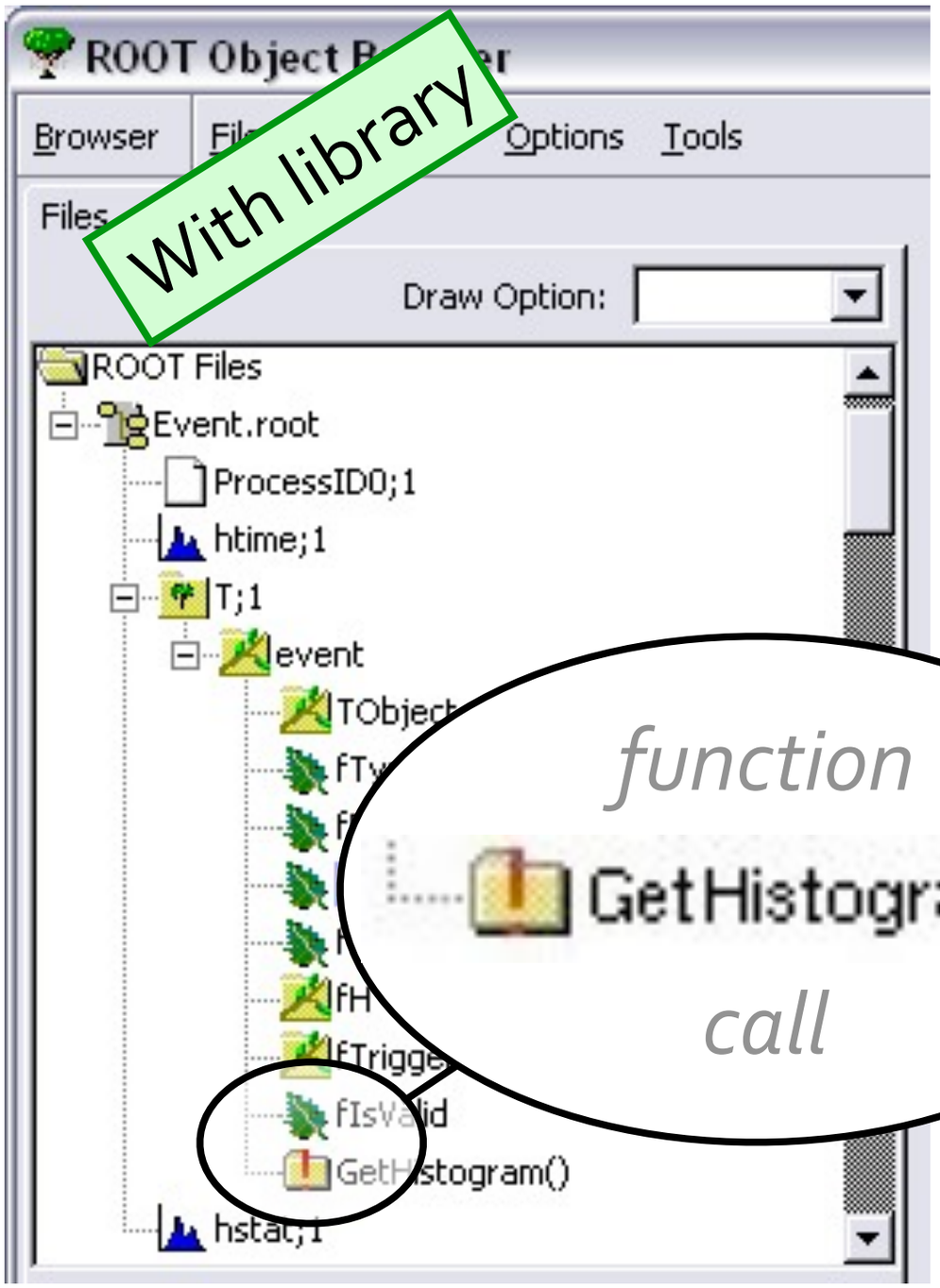
Files store reflection and data: need no library!

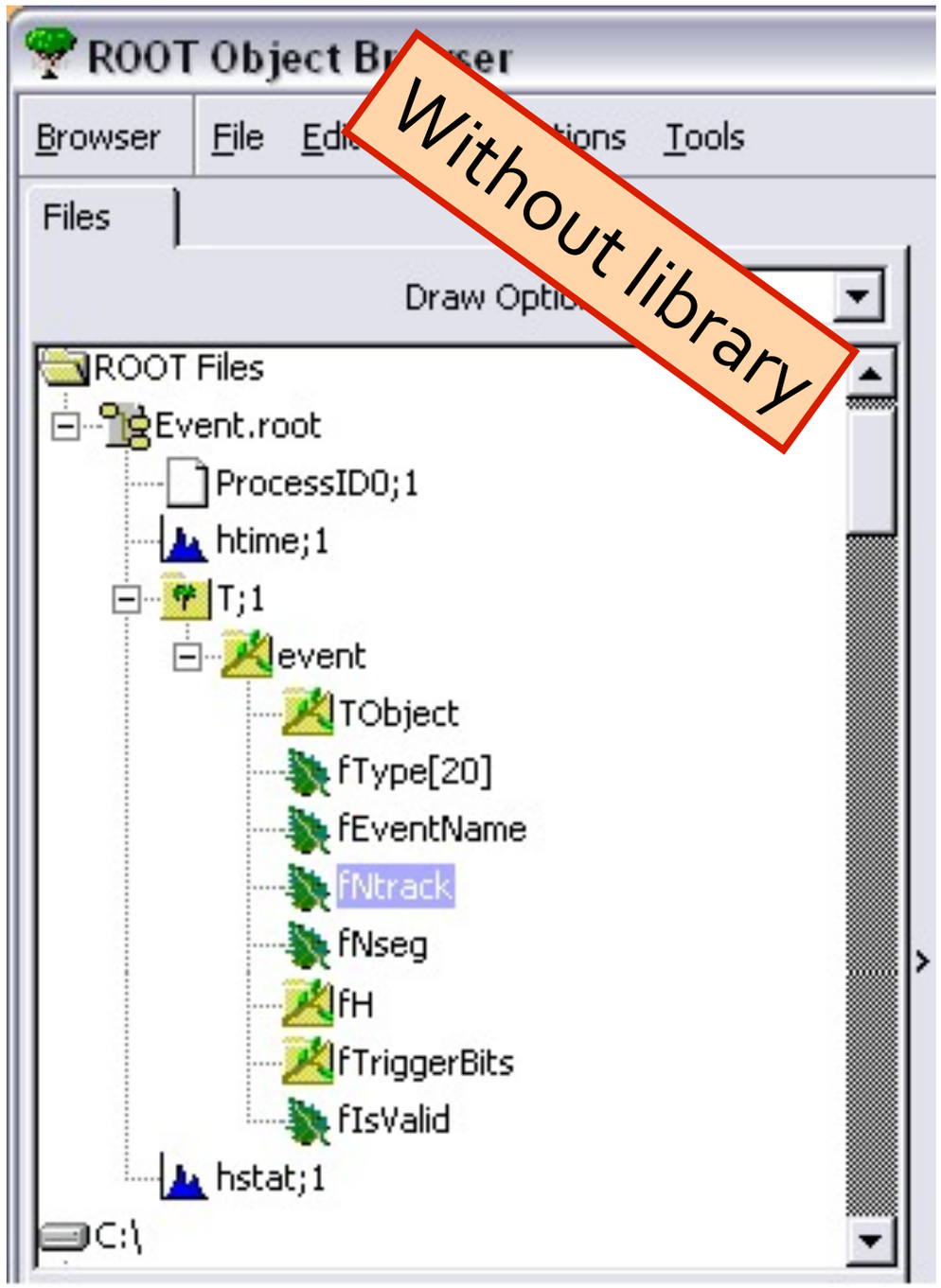




Reading Files

Files store reflection and data: need no library!





Retrieving Reflection Data

- When reflection data are not available, they can be retrieved from file
  - a dictionary library can also be generated
- Using TFile::MakeProject

```
TFile* f = TFile::Open("myfile.root");
f->MakeProject("mydir","*")
```

will create in the directory “mydir” heder files defining the classes present in the file. Only the data members can be retrieved not the member function

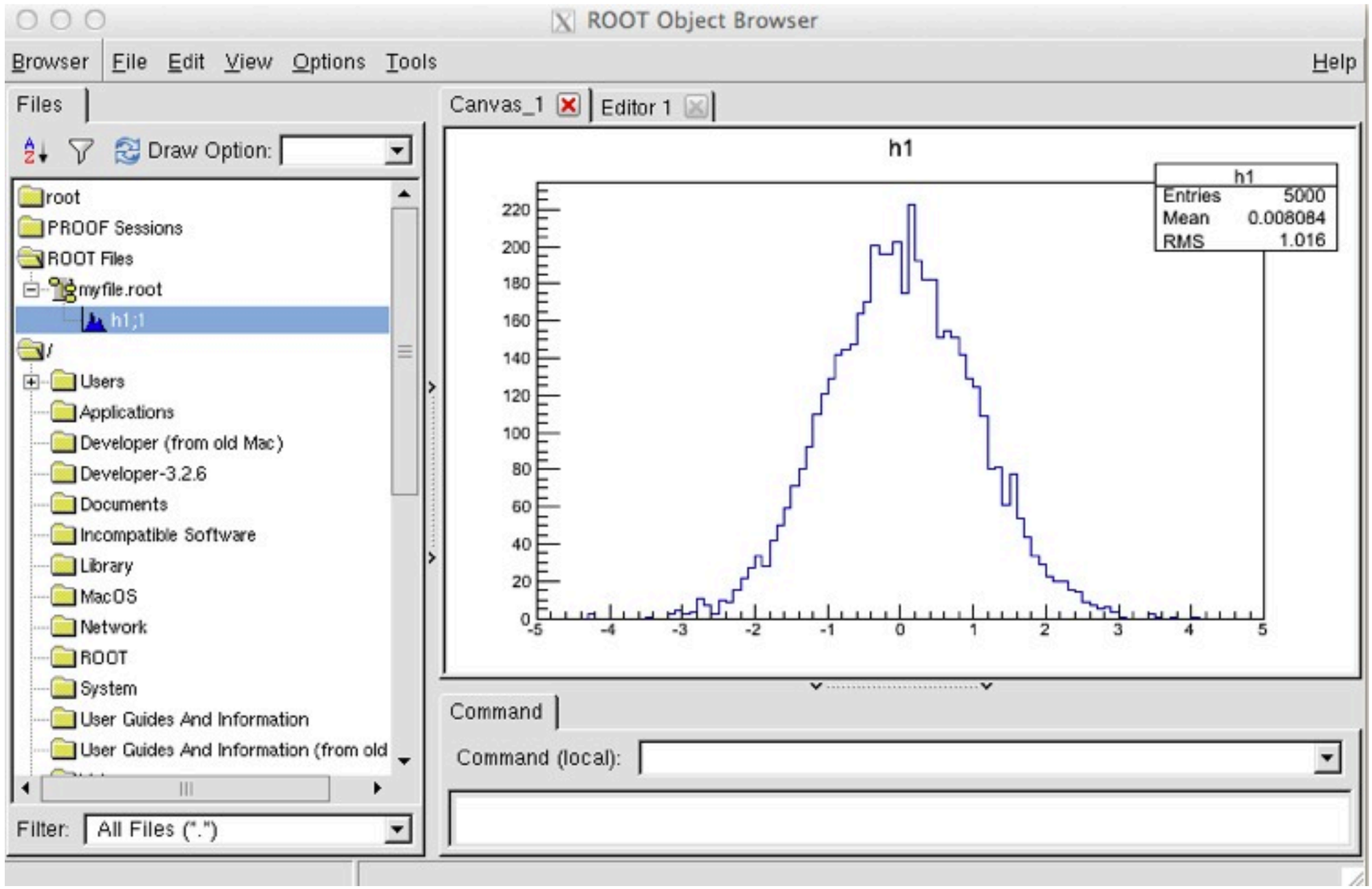
```
f->MakeProject("mydir","*","RECREATE++")
```

will generate and compile a dictionary library with the classes present in the file

TBrowser

- GUI for browsing ROOT objects written in a file

```
root [0] new TBrowser();
```



File Merging

- ROOT file containing the same data objects (e.g. histograms, Trees, etc...) can be merged using the command line tool $ROOTSYS/bin/hadd

```
$>  hadd  fileOut.root file1.root file2.root file3.root
```

```
$>  hadd  -h
Usage: hadd [-f[0-9]] [-k] [-T] [-O] [-n maxopenedfiles] [-v verbosity]
targetfile source1 [source2 source3 ...]
This program will add histograms from a list of root files and write them
to a target root file. The target file is newly created and must not
exist, or if -f ("force") is given, must not be one of the source files.
Supply at least two source files for this to make sense... ;-)
```

- hadd use functionality of TObject::Merge to merge the contained ROOT objects

ROOT Collection Classes

Collection Classes

ROOT collections polymorphic containers: hold pointers to `TObject`, so:

- Can only hold objects that inherit from `TObject`
- Return pointers to `TObject`, that have to be cast back to the correct subclass

```
void DrawHist(TObjArray *vect, int at)
{
   TH1F *hist = (TH1F*)vect->At(at);
   if (hist) hist->Draw();
}
```

TClonesArray

Array of objects of the same class ("clones")

Designed for repetitive data analysis tasks:

same type of objects created and deleted many times.

Standard Array

```
while (next_event()) {
  for (int i=0;i<N;++i)
    a[i] = new TTrack(x,y,z);
  do_something(a);
  a.clear();
}
```

objects created N x N-events times

TClonesArray

```
while (next_event()) {
  for (int i=0;i<N;++i)
    new(a[i]) TTrack(x,y,z);
  do_something(a);
  a.Delete();
}
```

object created only N times, re-used for each event

Traditional Arrays

Very large number of new and delete calls in large loops like this (N_{events} x N_{tracks} times new/delete):

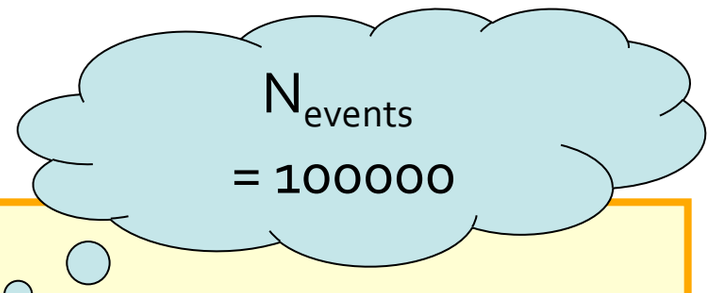
Traditional Arrays

Very large number of new and delete calls in large loops like this (N_{events} x N_{tracks} times new/delete):

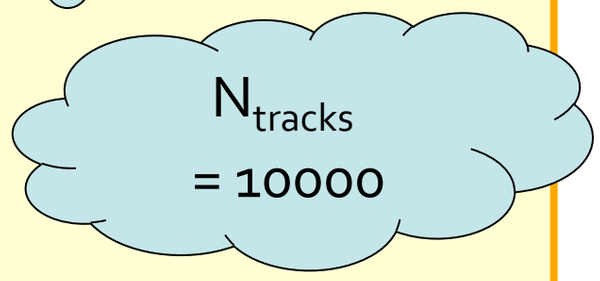
```
TObjArray a(10000);
while (TEvent *ev = (TEvent *)next()) {
   for (int i = 0; i < ev->Ntracks; ++i) {
      a[i] = new TTrack(x,y,z,...);
      ...
   }
   a.Delete();
}
```

Traditional Arrays

Very large number of new and delete calls in large loops like this (N_{events} x N_{tracks} times new/delete):




```
TObjArray a(10000);
while (TEvent *ev = (TEvent *)next()) {
   for (int i = 0; i < ev->Ntracks; ++i) {
      a[i] = new TTrack(x,y,z,...);
      ...
   }
   a.Delete();
}
```



Use of TClonesArray

You better use a `TClonesArray` which reduces the number of new/delete calls to only N_{tracks}:

N_{events} = 100000

```
TClonesArray a("TTrack", 10000);
while (TEvent *ev = (TEvent *)next()) {
   for (int i = 0; i < ev->Ntracks; ++i) {
      new(a[i]) TTrack(x,y,z,...);
      ...
   }
   a.Delete();
}
```

N_{tracks} = 10000

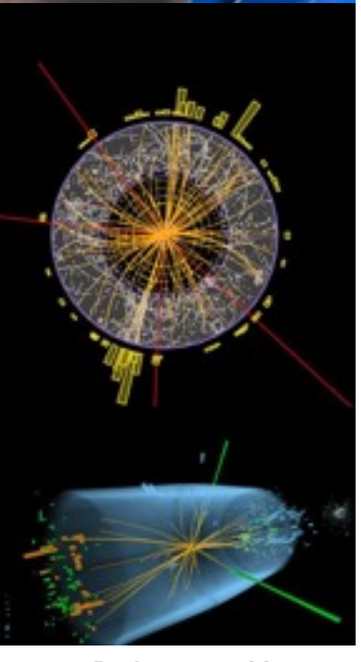
- Pair of new / delete calls cost about 4 µs
- Allocating / freeing memory N_{events}*N_{tracks} = 10^9 times costs about 1 hour!

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions and solve a simple exercises on ROOT I/O

ExerciseTwiki Page

ROOT Trees



- ROOT Trees:
  - `TNtuple` class (a simple Tree)
  - `TTree` class
- How to create a Tree and to write in a file
- How to read a Tree and query variables
- How to analyze the Tree
- Merging of Trees: `TChain`
- Using Tree Friends

Dr Lorenzo Moneta
CERN PH-SFT
CH-1211 Geneva 23
sftweb.cern.ch
root.cern.ch

Ntuple

- Ntuple class:
  - `TNtuple`
    - for storing tabular data
    - e.g. various rows with numbers

E	px	py	pz	pt	eta	phi
51.238284	25.706396	-0.238289	44.322524	25.7075	1.31298	-0.009269
61.68896	-21.06428	21.67916	53.775813	30.2273	1.34022	2.34181
69.232896	29.087514	21.827313	58.912468	36.3664	1.25952	0.643758
55.435615	-19.07243	-32.10133	40.973823	37.3397	0.948547	-2.10689
44.081717	-21.76992	18.725977	33.445573	28.7157	0.993186	2.43122
46.970421	-4.27962	-41.94243	-20.70599	42.1602	-0.473261	-1.67248
109.01623	14.823946	-24.63801	105.15587	28.7538	2.00801	-1.02915
81.517424	35.058621	-9.93301	-72.91995	36.4386	-1.44416	-0.27609
59.76741	-18.99337	-21.60084	52.390826	28.7636	1.3608	-2.29205
59.123105	-11.88863	56.536229	12.564104	57.7727	0.215796	1.77806
125.44969	30.001331	11.727174	-121.2436	32.2119	-2.03581	0.372627
44.246096	24.595778	-7.317789	36.044621	25.6613	1.14067	-0.289182
42.820008	29.287483	5.4862204	30.752201	29.7969	0.903863	0.185177
28.761648	-28.45137	-3.030433	-2.927228	28.6123	-0.102129	-3.03548
44.427016	34.281963	-11.5006	25.811686	36.1596	0.663957	-0.323673

ROOT Ntuple class

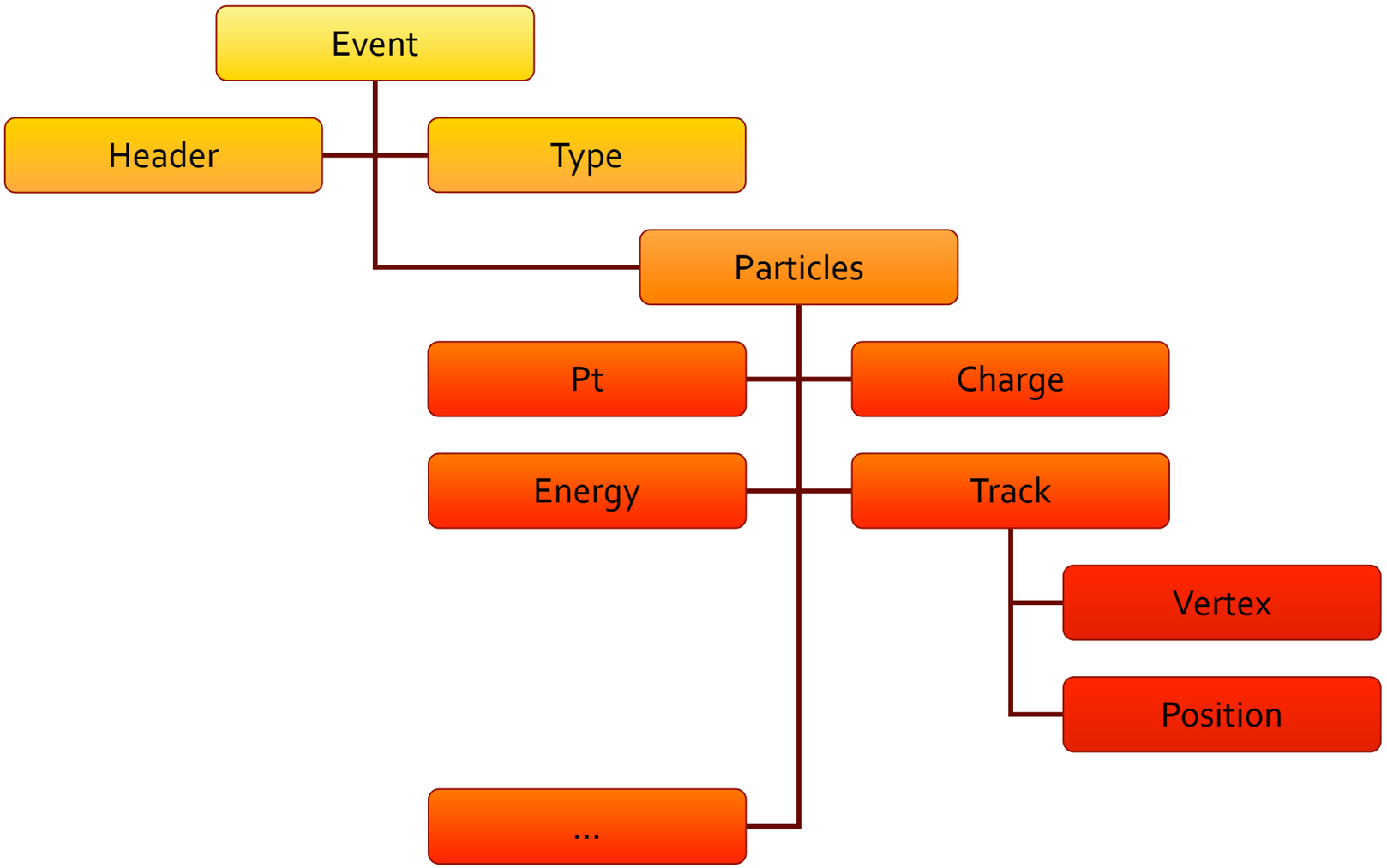
- ROOT N-tuple can store only floating point variables
  - limitation that all variables must be of the same type

```
TNtuple data("ntuple","Example N-tuple","x:y:z:t");

// fill it with random data
for (int i = 0; i<10000; ++i) {
    float x = gRandom->Uniform(-10,10);
    float y = gRandom->Uniform(-10,10);
    float z = gRandom->Gaus(0,5);
    float t = gRandom->Exp(10);

    data.Fill(x,y,z,t);
}
// write in a file
TFile f("ntuple_data.root","RECREATE");
data.Write();
f.Close();
```

  - not really useful for storing complex analysis data

ROOT Trees

- Tree class in ROOT
  - `TTree`
    - for storing complex data types (any type of objects)

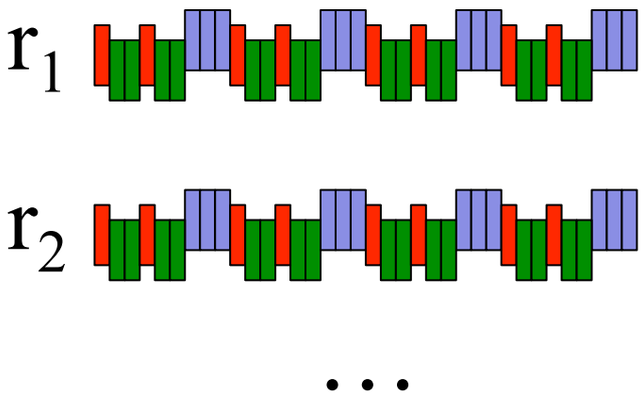


Why Trees ?

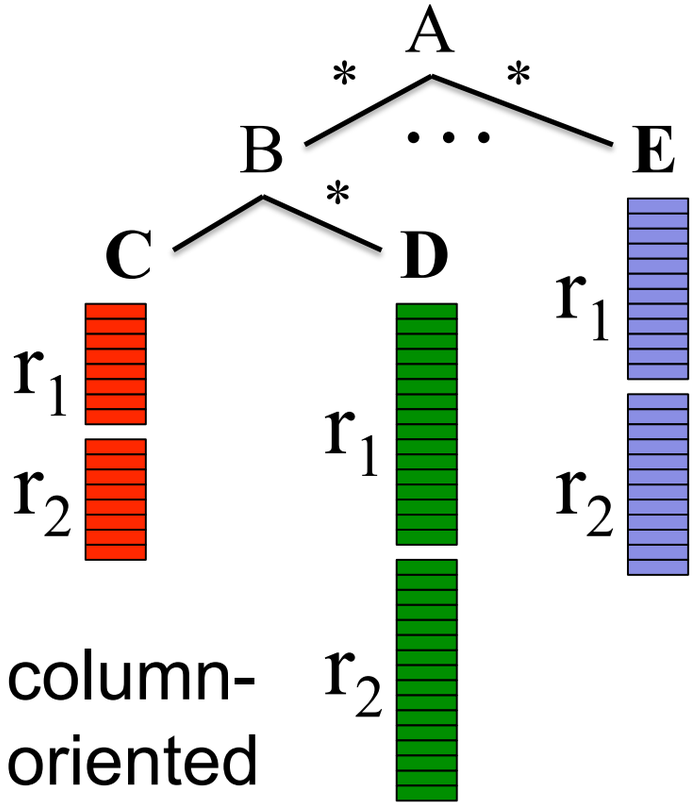
- `object.Write()` is convenient for simple objects like histograms, but inappropriate for saving collections of events containing complex objects
- When reading a collection (e.g. a TObjArray)
  - read all elements (all events) in memory
- With trees:
  - only a part of it (less I/O)
- Trees buffered to disk (TFile);
  - I/O is integral part of TTree concept
- Trees can read only a sub-set of all events
  - only the selected columns
  - Trees have a column oriented storage

Tree Access

- Databases have typically row wise access/storage
  - Can only access the full object (e.g. full event)
- ROOT trees have column wise access
  - Designed to access the object or a subset of the object attributes (e.g. only particles' energy)



record-oriented



column-oriented

Column vs Row Wise Access

- Advantage of column wise representation:
  - can read only interest part of event, e.g. read only the event muon candidates
    - Less I/O operation: → faster to read
  - same members consecutive, e.g. for object with position in X, Y, Z, and energy E, all X are consecutive, then come all Y, then Z, then E.
    - much higher compression efficiency: → less space on disk

- Disadvantage:
  - more expensive to write
    - adding new events to an existing tree
- ROOT Trees are designed to write once and to read many times

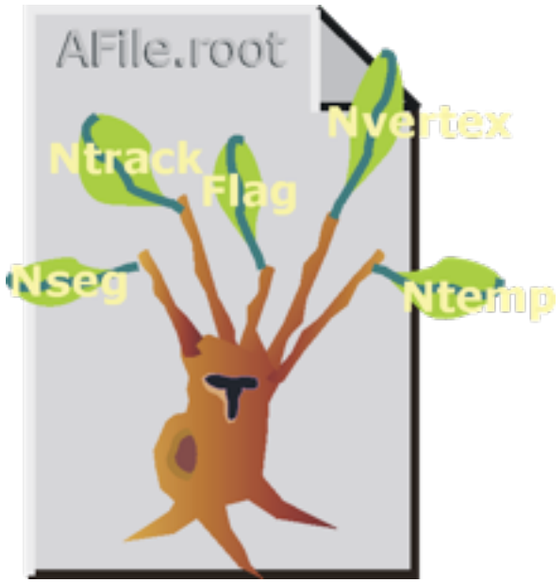
ROOT Trees Advantages

- ROOT tree class is designed for storing any type of object
- The ROOT Tree is
  - extremely efficient write once, read many.
  - Designed to store HEP events which have same data structure
    - size can be different but structure is the same
  - Trees allow fast direct and random access to any entry (sequential access is the best).
  - Optimized for network access (read-ahead).

ROOT Tree Structure: Branches

- A ROOT Tree is composed of Branches
  - a Branch (TBranch) can hold a simple variable, a list of variables, an object or even a collection of objects
    - no splitting: the whole object is written in the branch
    - splitting: the object member are assigned to separate branches

no splitting



splitting

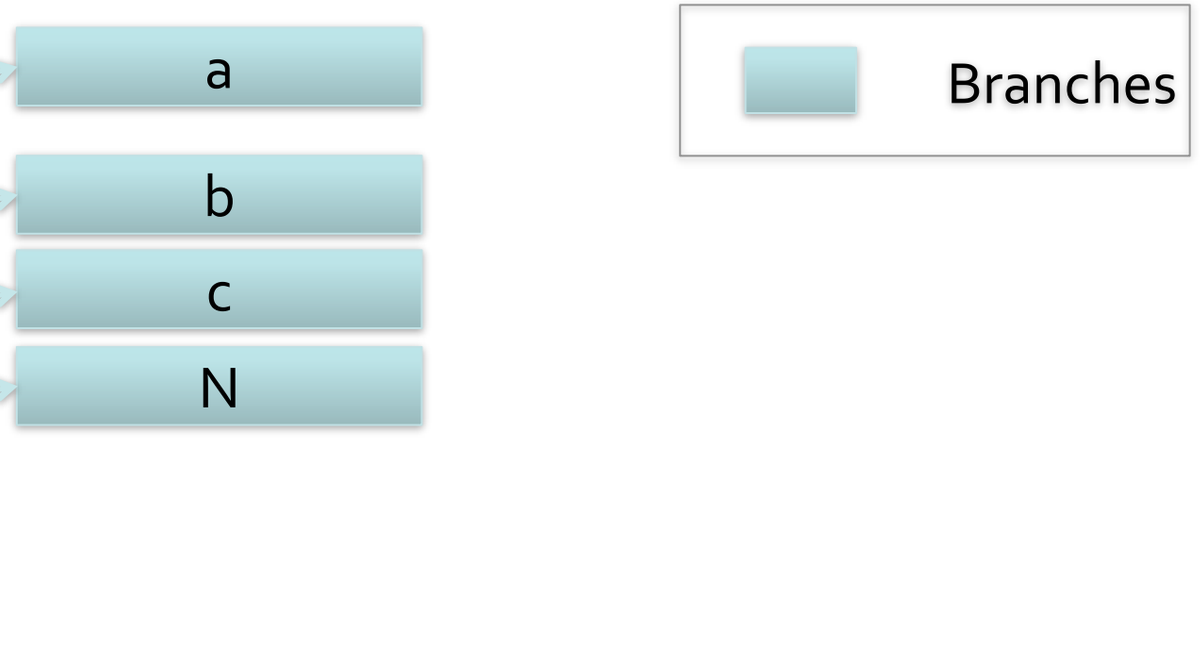
Branches and Leafs

- A Branch can contain another Branch
- The leaves (TLeaf) are the data containers of the branch
- It is possible to read only a sub-set of all the branches in a tree
  - variables or object known to be used together should be put in the same branch
    - faster read-access
- Branches of the same tree can also be written to separate files

Branch Creation from Class

Event Class

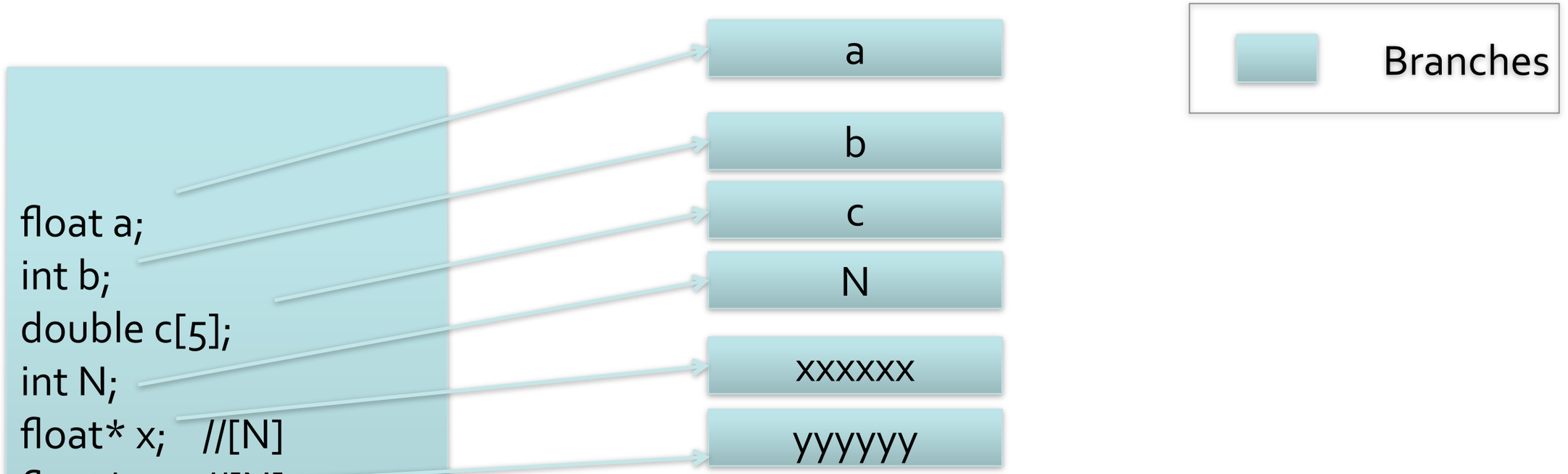
```
float a;
int b;
double c[5];
int N;
float* x;   //[N]
float* y;   //[N]
Class1 c1;
Class2 c2;   //!
Class3 *c3;
std::vector<T> v1;
std::vector<S*> v2;
TClonesArray *tc;
```



Branch Creation from Class

Event Class

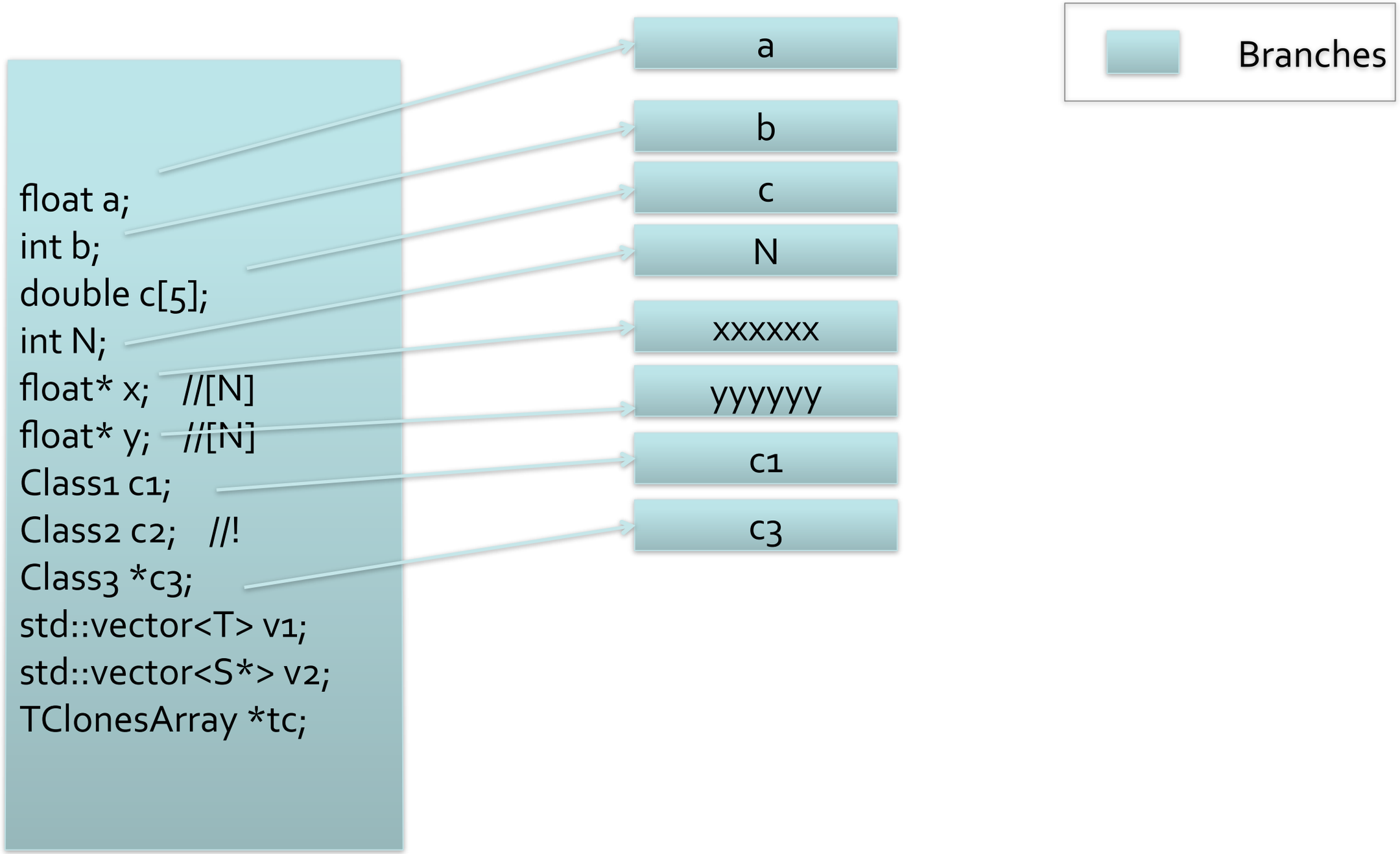
```
float a;
int b;
double c[5];
int N;
float* x;   //[N]
float* y;   //[N]
Class1 c1;
Class2 c2;   //!
Class3 *c3;
std::vector<T> v1;
std::vector<S*> v2;
TClonesArray *tc;
```



Branch Creation from Class

Event Class

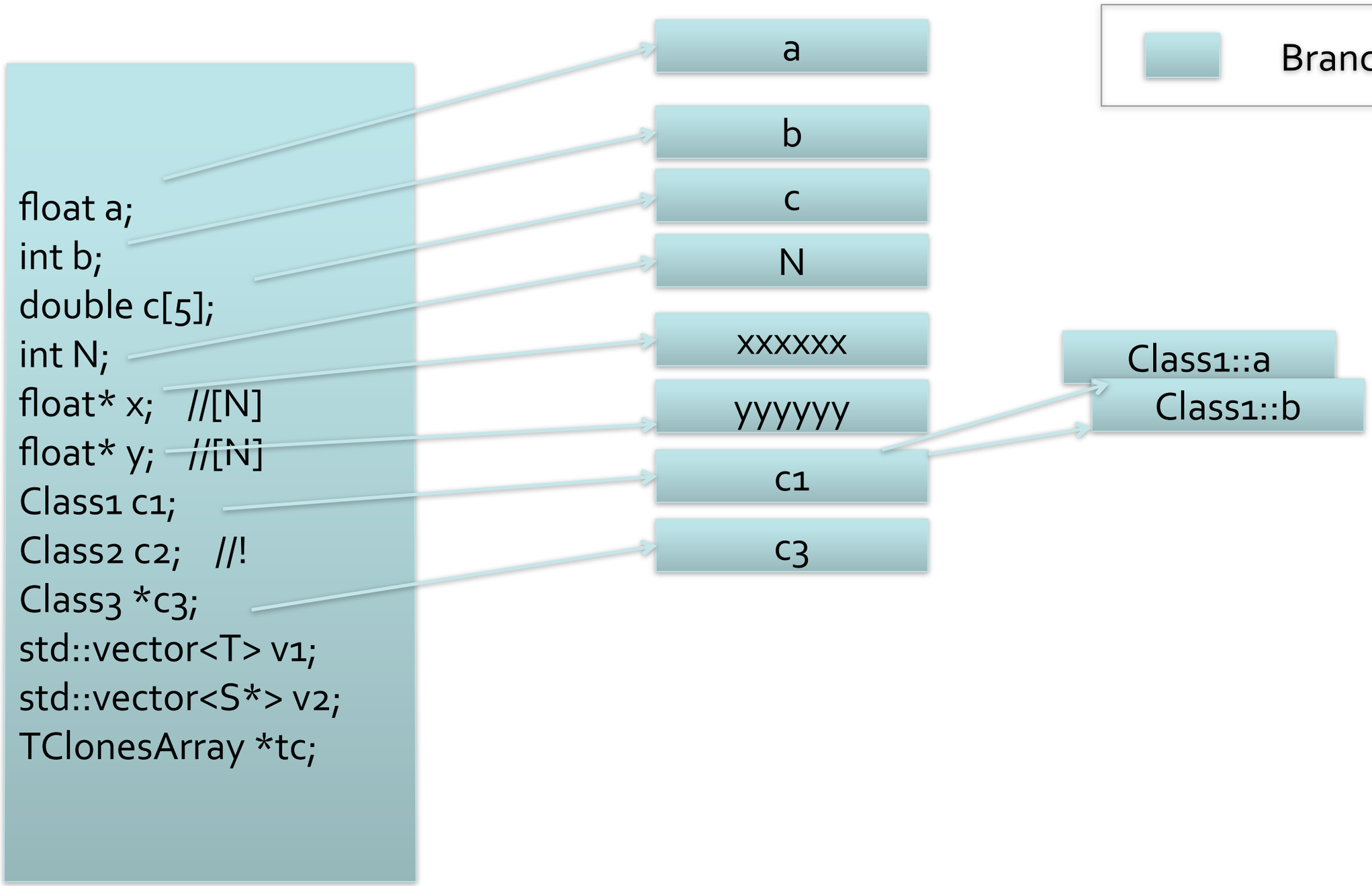
```
float a;
int b;
double c[5];
int N;
float* x;   //[N]
float* y;   //[N]
Class1 c1;
Class2 c2;   //!
Class3 *c3;
std::vector<T> v1;
std::vector<S*> v2;
TClonesArray *tc;
```



Branch Creation from Class

Event Class

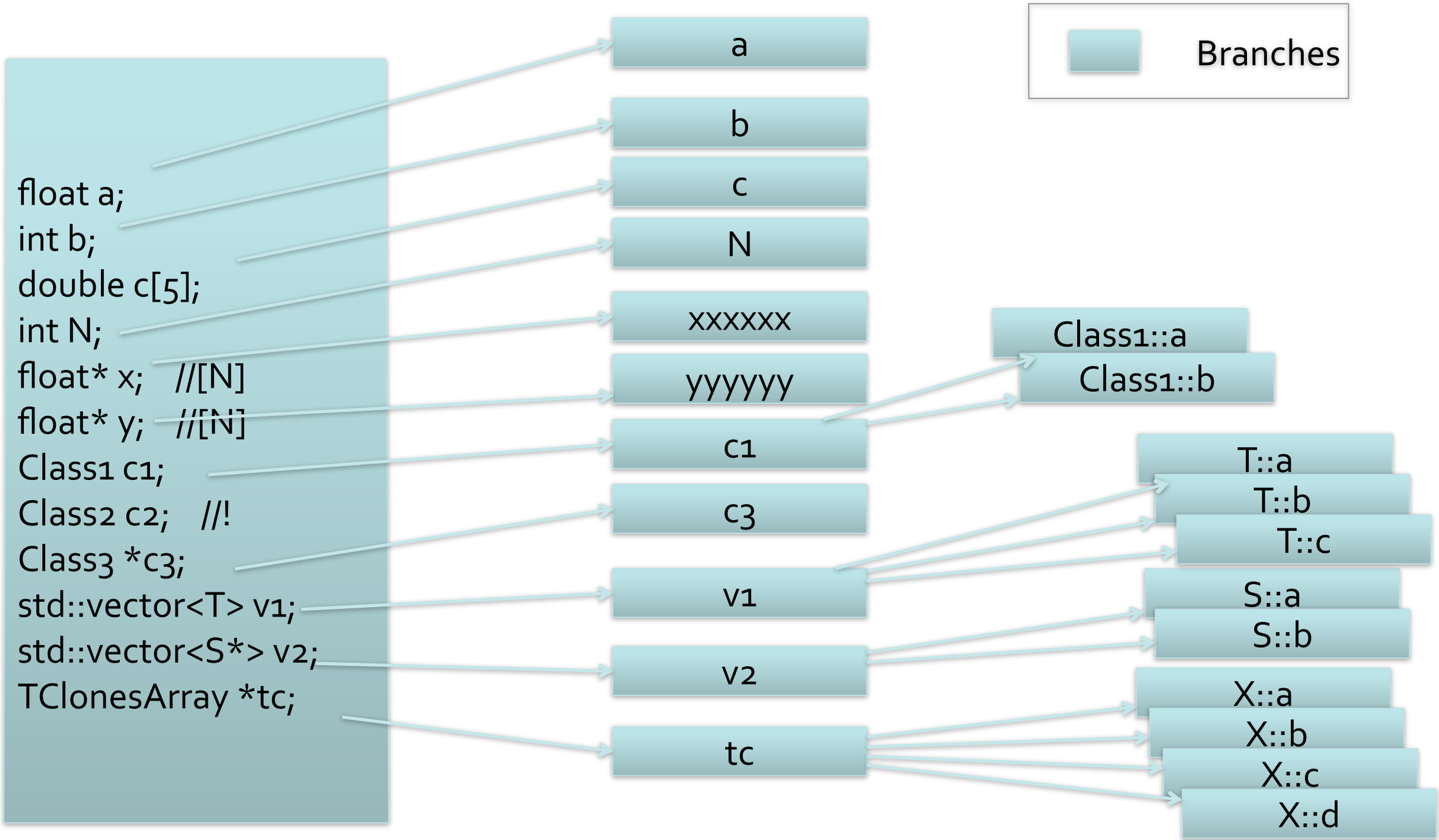
```
float a;
int b;
double c[5];
int N;
float* x;   //[N]
float* y;   //[N]
Class1 c1;
Class2 c2;   //!
Class3 *c3;
std::vector<T> v1;
std::vector<S*> v2;
TClonesArray *tc;
```



Branch Creation from Class

Event Class

```
float a;
int b;
double c[5];
int N;
float* x;   //[N]
float* y;   //[N]
Class1 c1;
Class2 c2;   //!
Class3 *c3;
std::vector<T> v1;
std::vector<S*> v2;
TClonesArray *tc;
```



ObjectWise/MemberWise

3 modes to stream an object

a
b
c
d

object-wise to a buffer

a1b1c1d1a2b2c2d2…anbncndn

ObjectWise/MemberWise

3 modes to stream an object

member-wise streaming of collections default since 5.27

a
b
c
d

object-wise to a buffer

member-wise to a buffer

a1b1c1d1a2b2c2d2...anbncndn

a1a2..anb1b2..bnc1c2..cnd1d2..dn

ObjectWise/MemberWise

3 modes to stream an object

member-wise streaming of collections default since 5.27

a
b
c
d

object-wise to a buffer

a1b1c1d1a2b2c2d2...anbncndn

member-wise to a buffer

a1a2..anb1b2..bnc1c2..cnd1d2..dn

each member to a buffer

a1a2...an

b1b2...bn

c1c2...cn

d1d2...dn

ObjectWise/MemberWise

3 modes to stream an object

member-wise streaming of collections default since 5.27

a
b
c
d

object-wise to a buffer

a1b1c1d1a2b2c2d2...anbncndn

member-wise to a buffer

a1a2..anb1b2..bnc1c2..cnd1d2..dn

each member to a buffer

a1a2...an

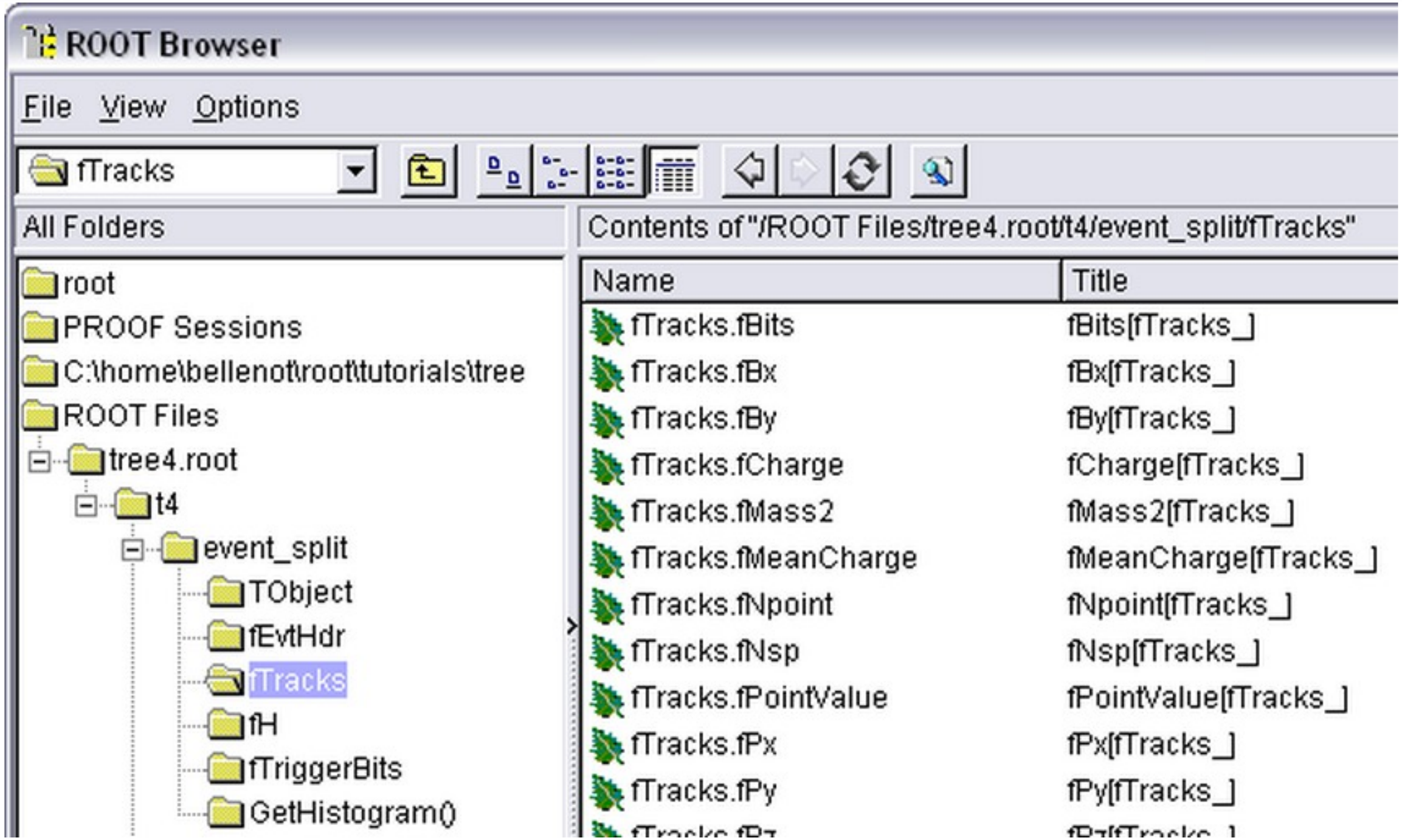
b1b2...bn

c1c2...cn

d1d2...dn

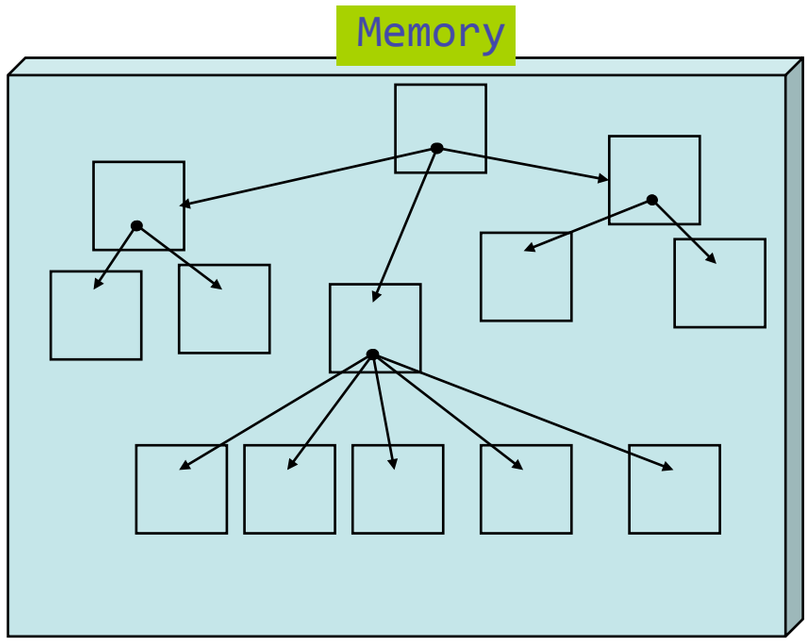
member-wise gives better compression

Tree structure



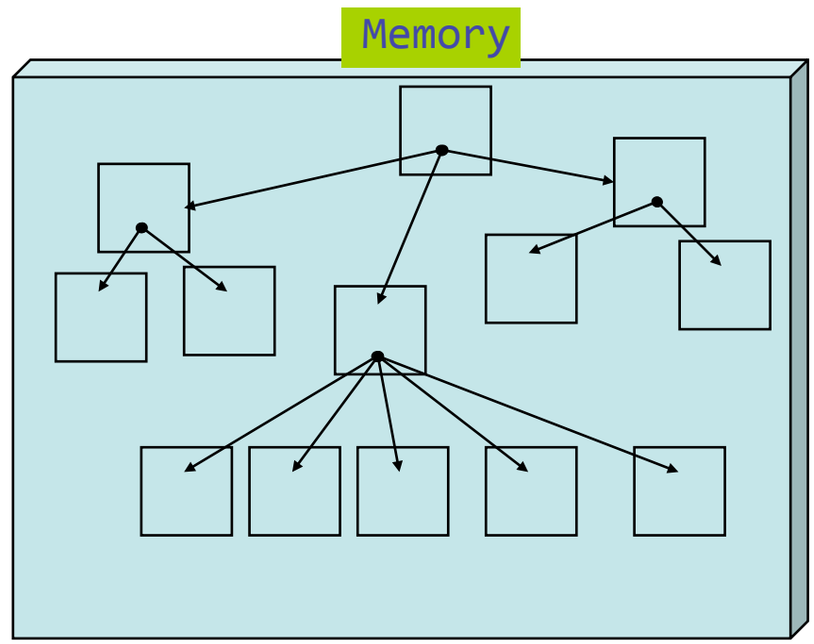
Memory ↔ Tree

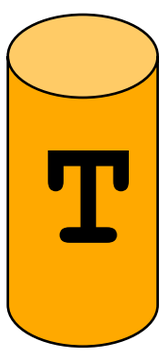
- Each Node is a branch in the Tree



Memory ↔ Tree

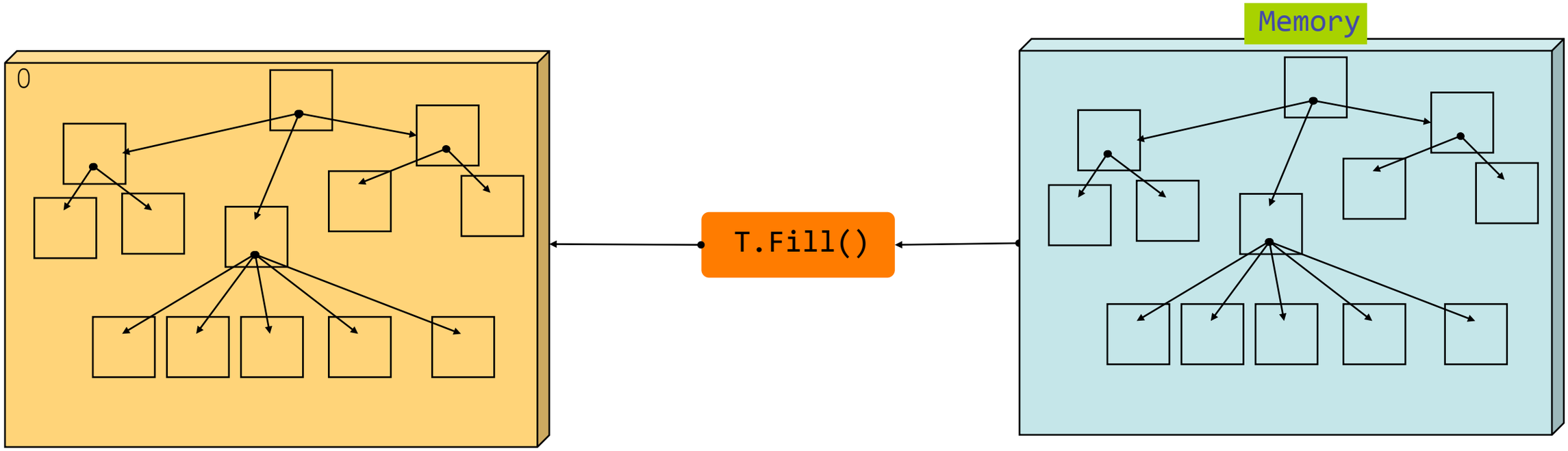
- Each Node is a branch in the Tree

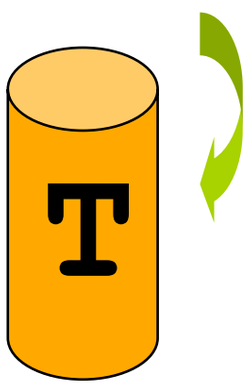




Memory ↔ Tree

- Each Node is a branch in the Tree



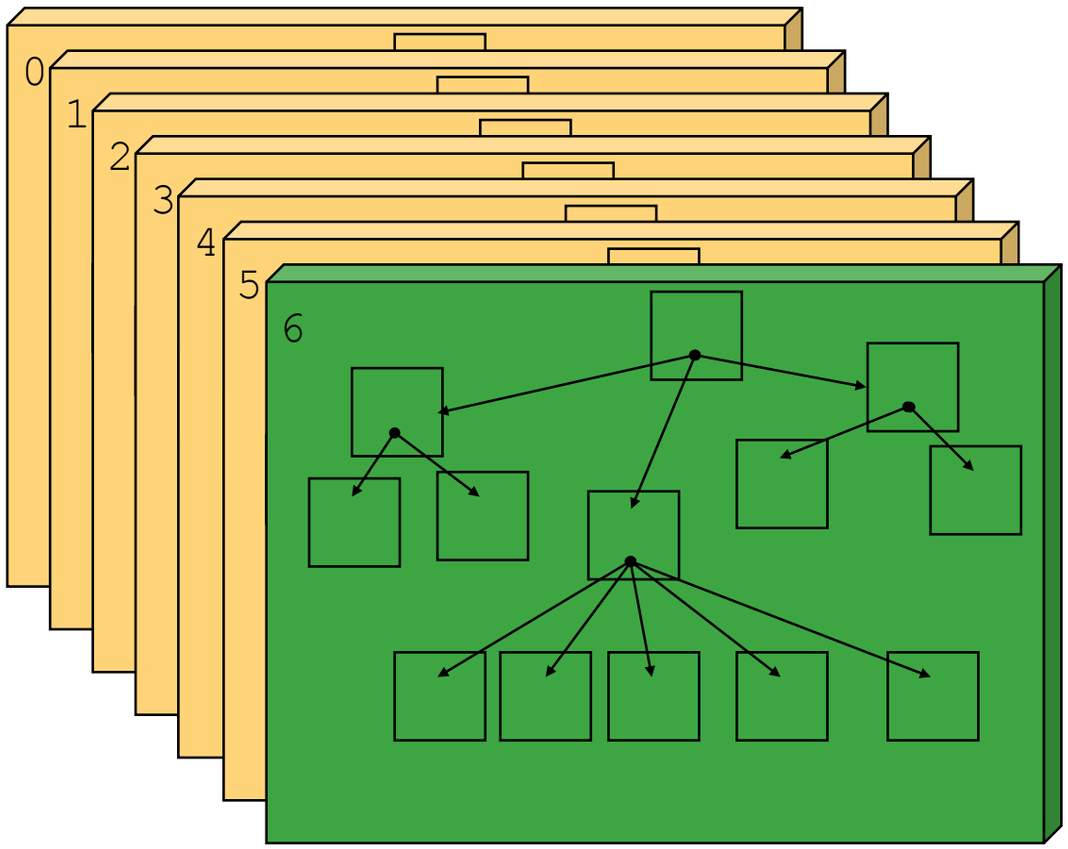


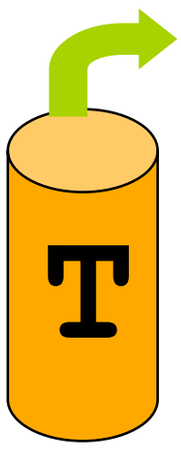
Memory ↔ Tree

- Each Node is a branch in the Tree

Memory ↔ Tree

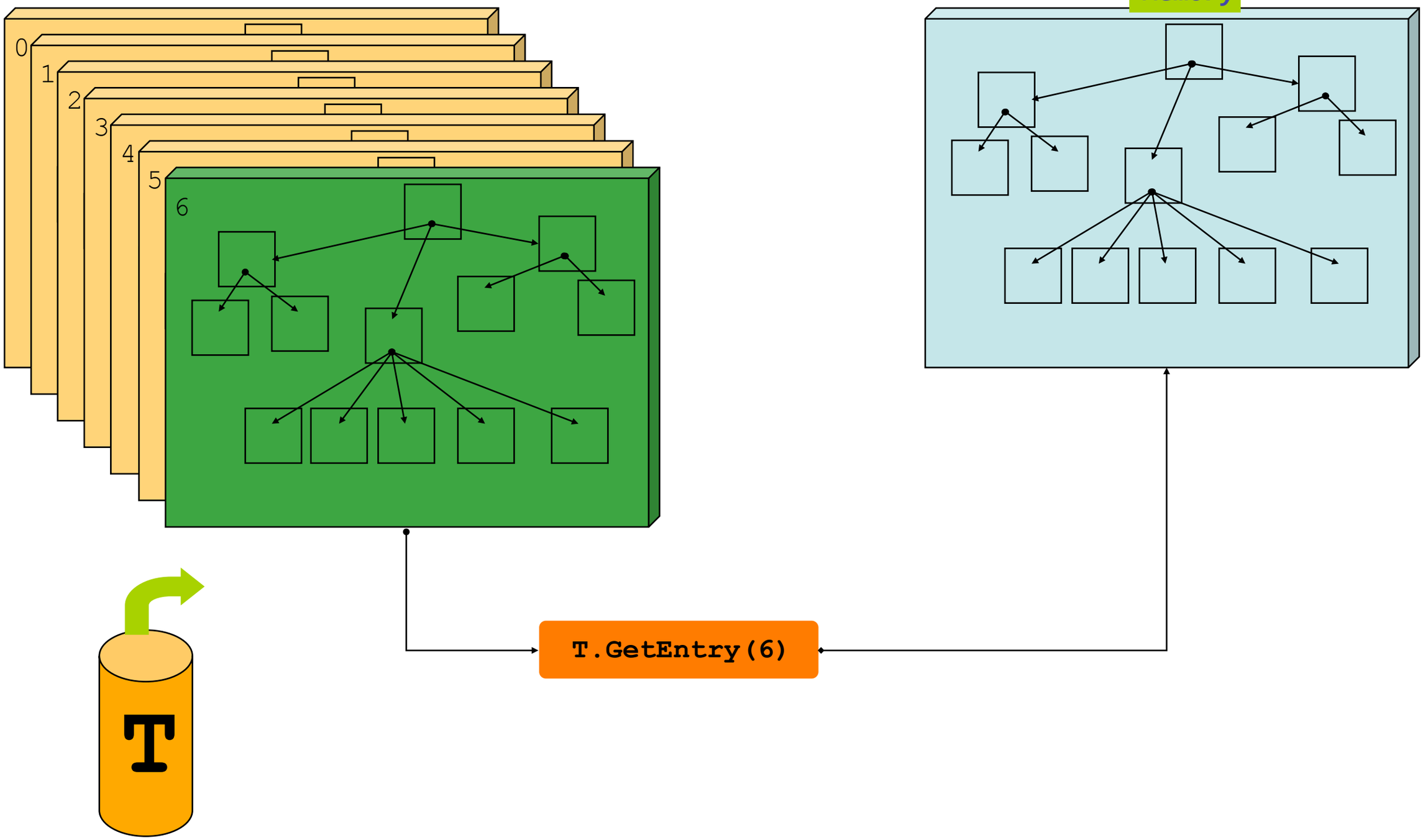
- Each Node is a branch in the Tree





Memory ↔ Tree

- Each Node is a branch in the Tree



5 Steps to Build a Tree

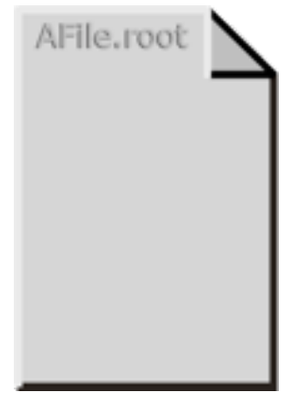
- Steps:

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

Building a ROOT Tree (1 and 2)

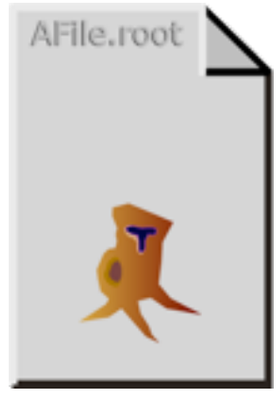
- Step 1:
  - Create a TFile class
    - Tree can be huge → need file for swapping filled entries

```
TFile *hfile = TFile::Open("AFile.root","RECREATE");
```



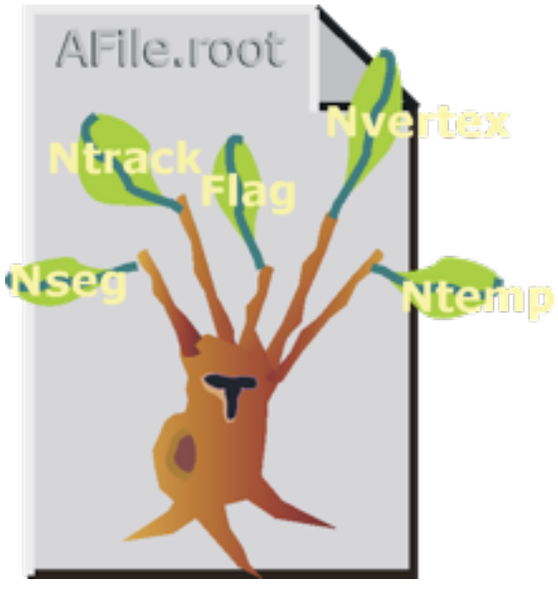
- Step 2:
  - Create a TTree class

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



Adding a Branch to the Tree

- Step 3: adding a branch. We need:
  - Name of the Branch (e.g. “eBranch”)
  - Address of the pointer to the object we want to store (e.g. Event **)
    - optionally we can specify also:
      - branch buffer size (default is 32000)
      - split level (default is 99, max splitting)



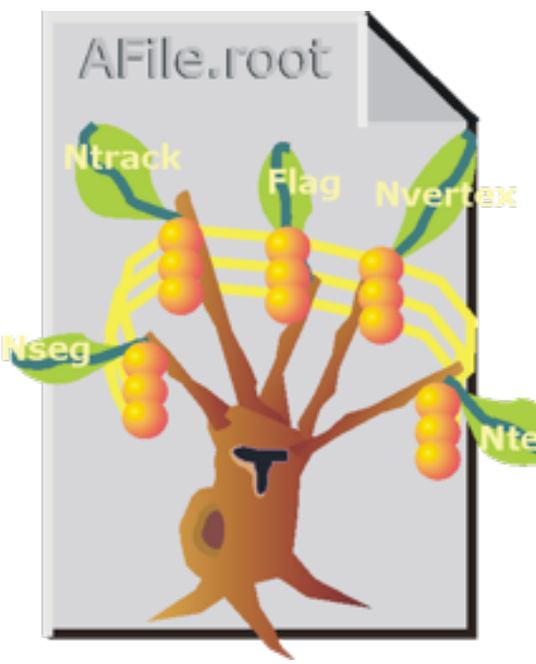

```
Event *myEvent = new Event();
myTree->Branch("eBranch", &myEvent);
```

`myEvent` is an hypothetical object of type `Event` we want to store in the tree

Note that we need to have generated the ROOT dictionary for the object we want to store

Fill the Tree

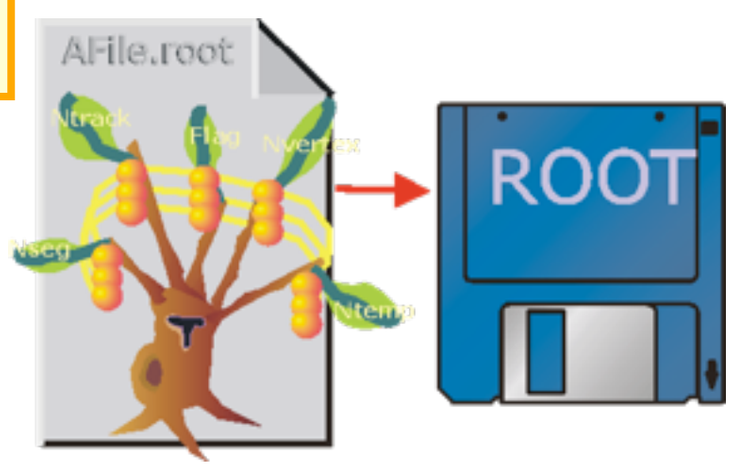
- Loop on the tree
- assign values to the object we want to store
  - e.g. by calling `myEvent->Generate`
- call `TTree::Fill()` creates a new entry in the tree:
  - snapshot of values of branches' objects




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

- After, write Tree to file:

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



Example Macro

- Example on how to create a TTree with the object "Event", fill with 10000 events and write to the file

```
void WriteTree()
{
   Event *myEvent = new Event();
   TFile f("AFile.root", "RECREATE");
   TTree *t = new TTree("myTree","A Tree");
   t->Branch("eBranch",&myEvent, 32000, 99);
   for (int e=0;e<100000;++e) {
      myEvent->Generate();  // hypothetical
      t->Fill();
   }
   t->Write();
}
```

Note: Event is an hypothetical class provided by the user
In TTree::Branch you can specify buffer size (32000) and split level (99)

Tree's with list of variables

- In case of a Tree containing a simple list of variables or array of variable, a variant exists:

```
void WriteTree()
{
   Int_t ntrack;
   Double_t p[100];
   TFile f("AFile.root", "RECREATE");
   TTree *t = new TTree("simpleTree","A Simple Tree");
   t->Branch("ntrack",&ntrack,"ntrack/I");
   t->Branch("p",p,"p[ntrack]/F");
   for (int e=0;e<100000;++e) {
      ntrack=...
      for (int i = 0; i < ntrack; ++i) p[i]=....
      t->Fill();
   }
   t->Write();
}
```

Reading a Tree

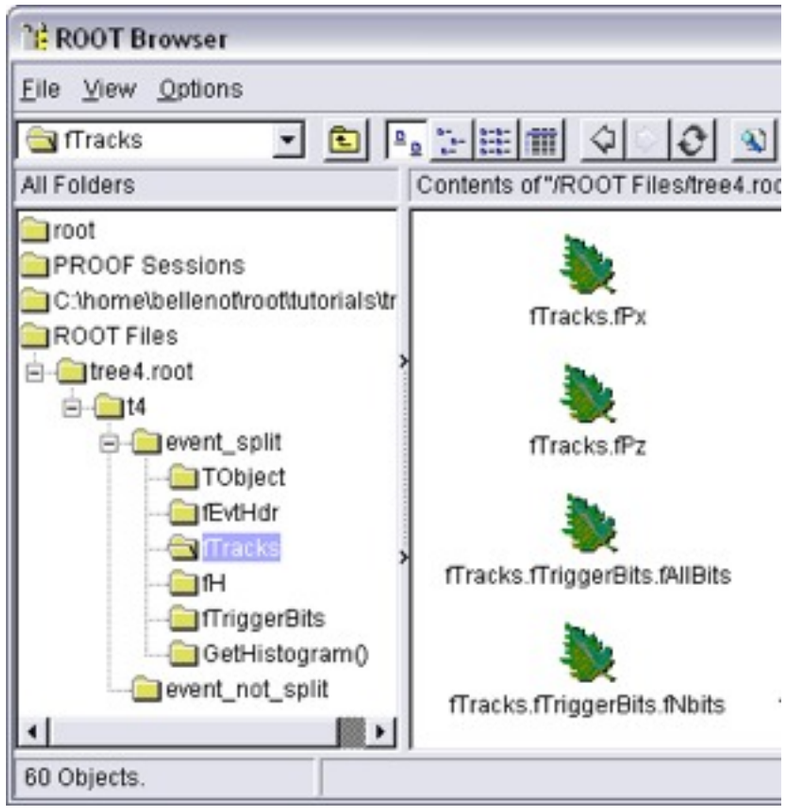
- Open the file and get the TTree object from the file

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

- Or browse the TTree using the TBrowser

- TTree::Print() shows the data layout
  - list of branches

- TTree::Draw("expression","selection") for drawing expression of variables



Scanning a Tree

- Syntax for querying a tree
  - Print the first 8 variables of the tree:

```
MyTree->Scan();
```

  - Prints all the variables of the tree:

```
MyTree->Scan("*");
```

  - Prints the values of var1, var2 and var3.

```
MyTree->Scan("var1:var2:var3");
```

  - A selection can be applied in the second argument:
  - Prints the values of var1, var2 and var3 for the entries where var1 is greater than 0

```
MyTree->Scan("var1:var2:var3", "var1>0");
```

- Use the same syntax as `TTree::Draw()`

Looking at the Tree

- More on scanning the Tree

```
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

- `TTree::Show(entry_number)` shows 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,…
```

How To Read a Tree in C++

- Create a variable pointing to the data (a pointer to data object)

```
Event * myEvent = 0;
```

- Associate a branch with the variable

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

- Read ith-entry in the Tree

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

- now variable points to data object for the i-th event

```
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
[...]
```

How To Read a Tree

- Example macro

```
void ReadTree() {
  TFile f("AFile.root");
  TTree *tree = (TTree*)f->Get("myTree");
  Event *myEvent = 0;
  TBranch* brEvent = 0;
  tree->SetBranchAddress("eBranch", &myEvent, &brEvent);
  Long64_t nent = tree->GetEntries();
  for (Long64_t i = 0; i < nbent; ++i) {
     tree->GetEntry(i);
     //brEvent->GetEntry(i); // to read only the branch
     myEvent->Analyze();
  }
```

- Data pointers (e.g. myEvent) MUST be set to 0
- SetBranchAddress requires address of pointers to event object and TBranch (i.e. Event**, TBranch **)

Accessing Tree Branches

- If we are interested in only some branches of a Tree:
  - Use `TTree::SetBranchStatus()` or just `TBranch::GetEntry()` to select the branches to be read
  - by default all branches are read when calling `TTree::GetEntry(event_number)`
  - Speed up considerably the reading phase
  - Example: reading only a branch with an array of muons

```
TClonesArray* myMuons = 0;
// disable all branches
tree->SetBranchStatus("*", 0);
// re-enable the "muon" branches
tree->SetBranchStatus("muon*", 1);
tree->SetBranchAddress("muon", &myMuons);
// now read (access) only the "muon" branches
for (Long64_t i = 0; i < myTree->GetEntries(); ++i) {
     tree->GetEntry(i);
```

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions and solve the exercises on creating, reading and analyzing the Tree

ExerciseTwiki Page

TChain: The Forest

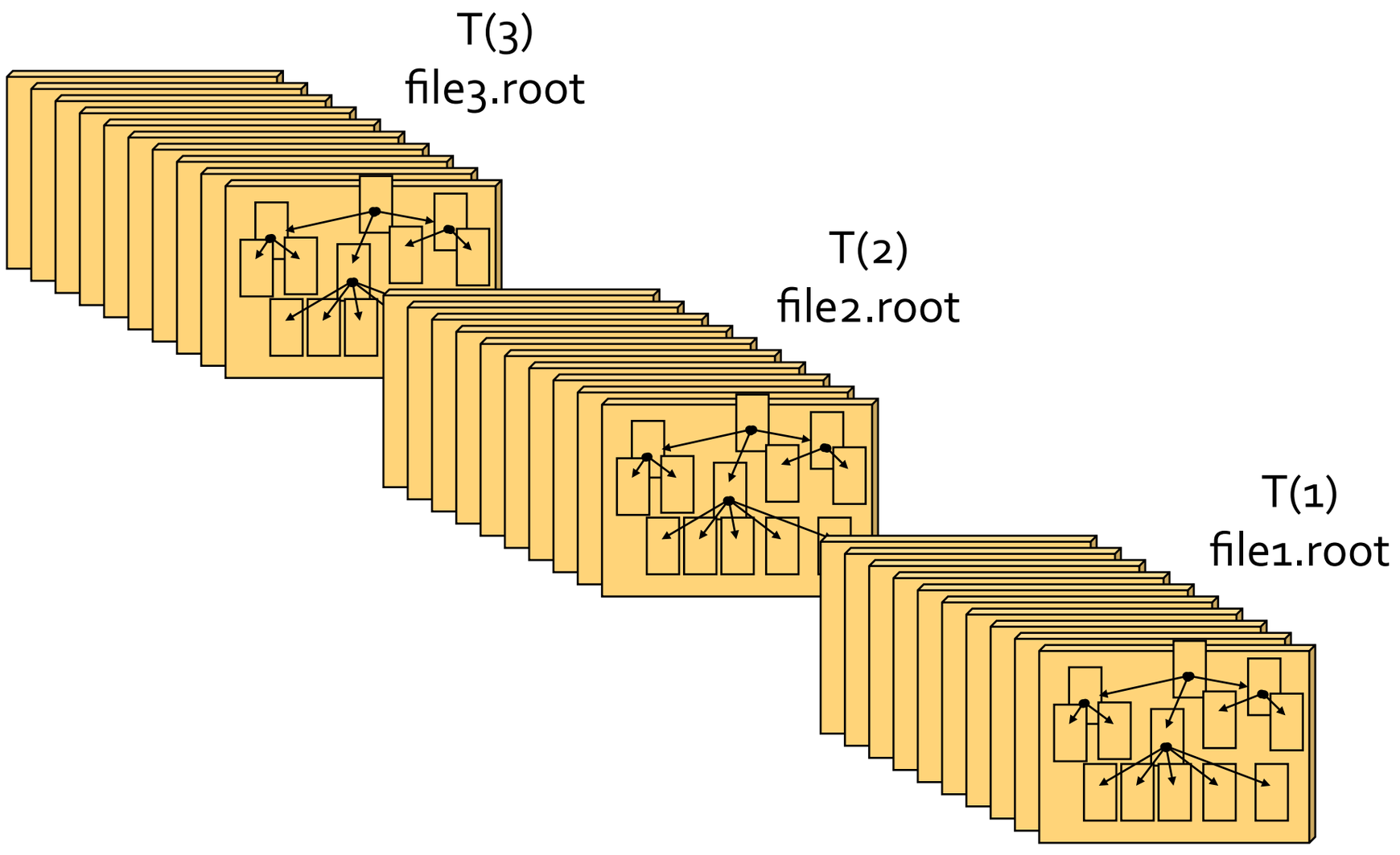
- Collection of Trees:
  - list of ROOT files containing the same tree
- Same semantics as `TTree`.
  - As an example, assume we have three files called file1.root, file2.root, file3.root. Each contains tree called "T". Create a chain:

```
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`!

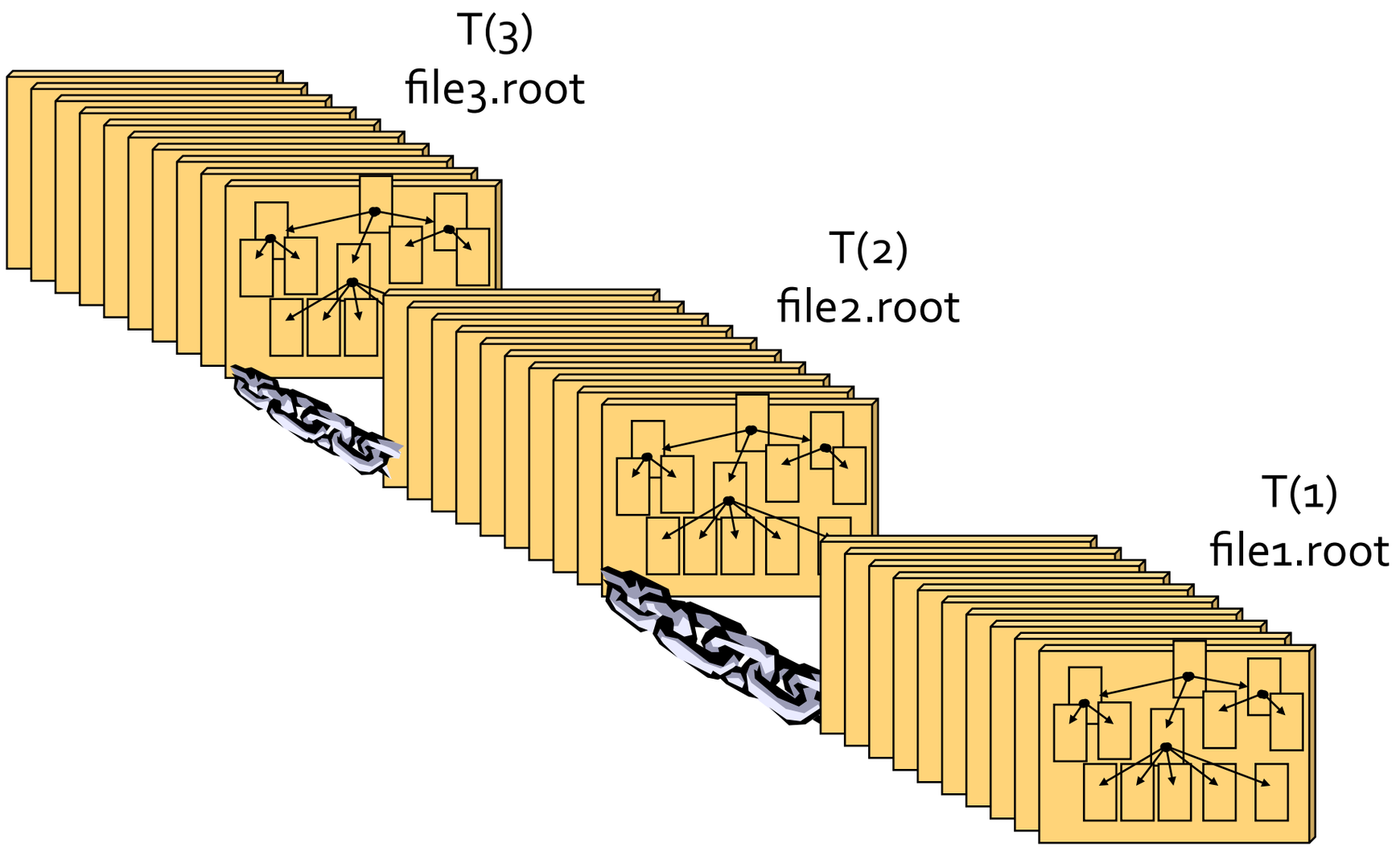
TChain

- Chain Files together



TChain

- Chain Files together



Analysis of TTree

- Different ways of analyzing trees:
  - inspection of variables:
    - use TTree::Draw() which can be extended using a function defined in a C macro (see TTree::MakeProxy)
  - write your own C++ code
    - require declaring and setting address for branches
    - ROOT provides a facility for creating some skeleton analysis code to read and loop a Tree
      - TTree::MakeClass
    - User still control iterations on TTree
- Using TSelector
  - ROOT controls iterations
    - can be parallelized using PROOF

TTree in Analysis (TTree::Draw)

- TTree::Draw for interactive queries of a Tree
  - suppose we have a tree with a branch “tracks” containing a `std::vector<ROOT::Math::XYZTVector>`

```
*******************************************************************************
*Br    0 :tracks    : Int_t tracks_                                              *
*Entries :    10000 : Total  Size=     103261 bytes  File Size  =      28261 *
*Baskets :        5 : Basket Size=      32000 bytes  Compression=   2.84     *
*.............................................................................*
*Br    1 :tracks.fCoordinates.fX : Double_t fX[tracks_]                         *
*Entries :    10000 : Total  Size=    8079269 bytes  File Size  =    7819412 *
*Baskets :      249 : Basket Size=    3990016 bytes  Compression=   1.03     *
*.............................................................................*
*Br    2 :tracks.fCoordinates.fY : Double_t fY[tracks_]                         *
*Entries :    10000 : Total  Size=    8079269 bytes  File Size  =    7819897 *
*Baskets :      249 : Basket Size=    3990016 bytes  Compression=   1.03     *
*.............................................................................*
*Br    3 :tracks.fCoordinates.fZ : Double_t fZ[tracks_]                         *
*Entries :    10000 : Total  Size=    8079269 bytes  File Size  =    7786816 *
*Baskets :      249 : Basket Size=    3990016 bytes  Compression=   1.04     *
*.............................................................................*
*Br    4 :tracks.fCoordinates.fT : Double_t fT[tracks_]                         *
*Entries :    10000 : Total  Size=    8079269 bytes  File Size  =    7663469 *
*Baskets :      249 : Basket Size=    3990016 bytes  Compression=   1.05     *
*.............................................................................*
```

TTree::Draw syntax

`TTree::Draw`(“expression”, “selection(weight)”)

- draw X component of all tracks

```
tree->Draw(“tracks.fX”);
```

- draw Eta of all tracks

```
tree->Draw(“tracks.Eta()”);
```

- draw Eta of tracks with pt > 5

```
tree->Draw(“tracks.Eta()”,”tracks.Pt()> 5”);
```

- draw number of tracks

```
tree->Draw(“@tracks.size()”);
```

  - note special symbol “@” to access collection object

TTree::Draw syntax (2)

- draw Pt of first track

```
tree->Draw("tracks[0].Pt");
tree->Draw("@tracks.front().Pt()""
```

- draw Px vs Py for all tracks

```
tree->Draw("tracks.X():tracks.Y()","","colz");
```

  - note we passed a graphics option colz for the histogram

- draw P vs Eta in a TProfile plot with 30 bins [-3,3]

```
tree->Draw("tracks.Pt()::tracks.Eta() >> ph(30,-3,3)",
           "","prof");
```

- see more in TTree::Draw documentation

TTree::Draw

- `TTree::Draw` is powerful and can make queries on variable of a tree and function of variables
- can call simple member functions of objects
  - member functions with void arguments or taking values
- cannot call member functions having objects as arguments
  - e.g. this does not work !

```
tree->Draw("(tracks[0]+tracks[1]).M()");
```

- Solution for more complex interactive analysis:
  - write your own function in C++ code

Special TTree::Draw functions

- These functions can be used to build `TTree::Draw` expressions:
  - Entry$ return the tree entry number

```
tree->Draw("Entry$");
```

  - Length$(formula) : return the total number of element

```
tree->Draw("Length$(tracks)");
```

  - Sum$(formula) : return the total sum of element

```
tree->Draw("Sum$(tracks)");
```

- More functions are available, see `TTree::Draw` documentation

Using a Proxy function

- Create a macro `proxy.C`

```
TH1F *h1;

void proxy_Begin(TTree*) {
  h1 = new TH1F("h1","Invariant Mass",100,0,100);
}

double proxy() {
   h1->Fill ( (tracks[0] + tracks[1]).M() );
   return 0;
}

void proxy_Terminate() {
  h1->Draw();
}
```

- use macro in `TTree::Draw`

```
tree->Draw("proxy.C");
```

  - any user selection can be added in the `proxy()` function

Using Make Class

```
root[1] tree->MakeClass("MyClass");
```

- will generate a MyClass.h and MyClass.C files with the skeleton code for doing analysis
  - declarations for all tree branches
  - setting the corresponding branch address
- After having filled the functions MyClass::Loop with the needed analysis code, run on the tree data:

```
root[2] .L MyClass.C
root[3] MyClass myclass;
root[4] myclass.Loop();
```

Example MyClass.h

```
class MyClass {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain

   // Declaration of leaf types
   vector<ROOT::Math::LorentzVector<ROOT::Math::PxPyPzE4D<double> > > *tracks;

   // List of branches
   TBranch        *b_tracks;   //!

   MyClass(TTree *tree=0);
   virtual ~MyClass();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};
```

NOTE: To have correct branch top level definition, branches must be not splitted

Example MyClass.h

This is what you get with split branch

```
class MyClass {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain

   // Declaration of leaf types
   Int_t           tracks_;
   Double_t        tracks_fCoordinates_fX[kMaxtracks];   //[tracks_]
   Double_t        tracks_fCoordinates_fY[kMaxtracks];   //[tracks_]
   Double_t        tracks_fCoordinates_fZ[kMaxtracks];   //[tracks_]
   Double_t        tracks_fCoordinates_fT[kMaxtracks];   //[tracks_]

   // List of branches
   TBranch        *b_tracks_;   //!
   TBranch        *b_tracks_fCoordinates_fX;   //!
   TBranch        *b_tracks_fCoordinates_fY;   //!
   TBranch        *b_tracks_fCoordinates_fZ;   //!
   TBranch        *b_tracks_fCoordinates_fT;   //!

   MyClass(TTree *tree=0);
   virtual ~MyClass();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};
```

Example MyClass.C

- Fill in Loop() the user code for analysis
  - e.g. plot invariant mass of tracks

```
void MyClass::Loop()
{
  Long64_t nentries = fChain->GetEntriesFast();

   TH1D * h1 = new TH1D("h1","Invariant Mass of all tracks", 100, 0,100);

   Long64_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      //fChain->GetEntry(jentry);
      b_tracks->GetEntry(jentry);    // faster to read only the branch
      // if (Cut(ientry) < 0) continue;

      for (unsigned int i = 0; i < (*tracks).size() ; ++i)
         for (unsigned int j = i+1; j < (*tracks).size() ; ++j)
            h1->Fill( ( (*tracks)[i]+(*tracks)[j] ).M() );
   }
   h1->Draw();
}
```

TSelector

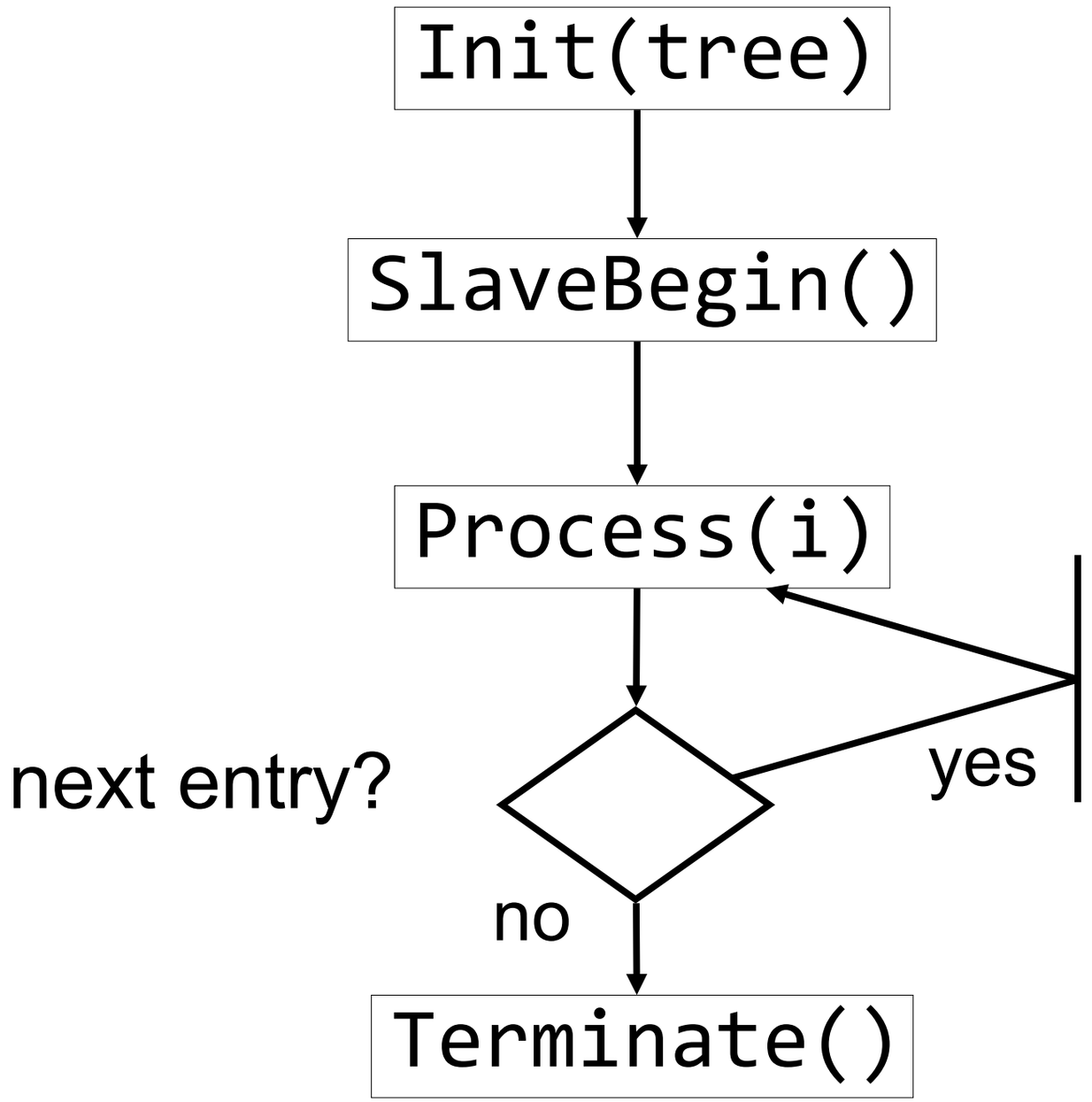
- Another way to analyze Tree is using the TSelector class
  - the user creates a new class `MySelector` deriving from **`TSelector`**

```
root[1] tree->MakeSelector("MySelector");
```

    - generates file `MySelector.h` and `MySelector.C`
  - the `MySelector` object is used in `TTree::Process(TSelector*,...)`
  - ROOT invokes the `TSelector's` functions which are virtuals, so the user provided function implemented in `MySelector` will be called.

Tree Data Access

E.g. `tree->Process("MySelector.C+")`



TSelector

TSelector

Steps of ROOT using a `TSelector`:

TSelector

Steps of ROOT using a `TSelector`:

1. ***setup*** **`TMySelector::Init(TTree *tree)`**
   `fChain = tree; fChain->SetBranchAddress()`
   initialize branches

TSelector

Steps of ROOT using a `TSelector`:

1. ***setup*** **`TMySelector::Init(TTree *tree)`**
   `fChain = tree; fChain->SetBranchAddress()`
   initialize branches

2. ***start*** **`TMySelector::SlaveBegin()`**
   create histograms

TSelector

Steps of ROOT using a `TSelector`:

1. ***setup*** **`TMySelector::Init(TTree *tree)`**
   `fChain = tree; fChain->SetBranchAddress()`
   initialize branches

2. ***start*** **`TMySelector::SlaveBegin()`**
   create histograms

3. ***run*** **`TMySelector::Process(Long64_t)`**
   `fChain->GetTree()->GetEntry(entry);`
   analyze data, fill histograms,…

TSelector

Steps of ROOT using a `TSelector`:

1. ***setup*** **`TMySelector::Init(TTree *tree)`**
   `fChain = tree; fChain->SetBranchAddress()`
   initialize branches

2. ***start*** **`TMySelector::SlaveBegin()`**
   create histograms

3. ***run*** **`TMySelector::Process(Long64_t)`**
   `fChain->GetTree()->GetEntry(entry);`
   analyze data, fill histograms,…

4. ***end*** **`TMySelector::Terminate()`**
   fit histograms, write them to files,…

TTReeReader

- New functionality to read TTree in ROOT 6

```
void TreeReaderSimple() {
   TH1F *myHist = new TH1F("h1","ntuple",100,-4,4);

   TFile *myFile = TFile::Open("hsimple.root");
   TTreeReader myReader("ntuple", myFile);

   TTreeReaderValue<Float_t> myPx(myReader, "px");
   TTreeReaderValue<Float_t> myPy(myReader, "py");

   while (myReader.Next()) {
      myHist->Fill(*myPx + *myPy);
   }

   myHist->Draw();
}
```

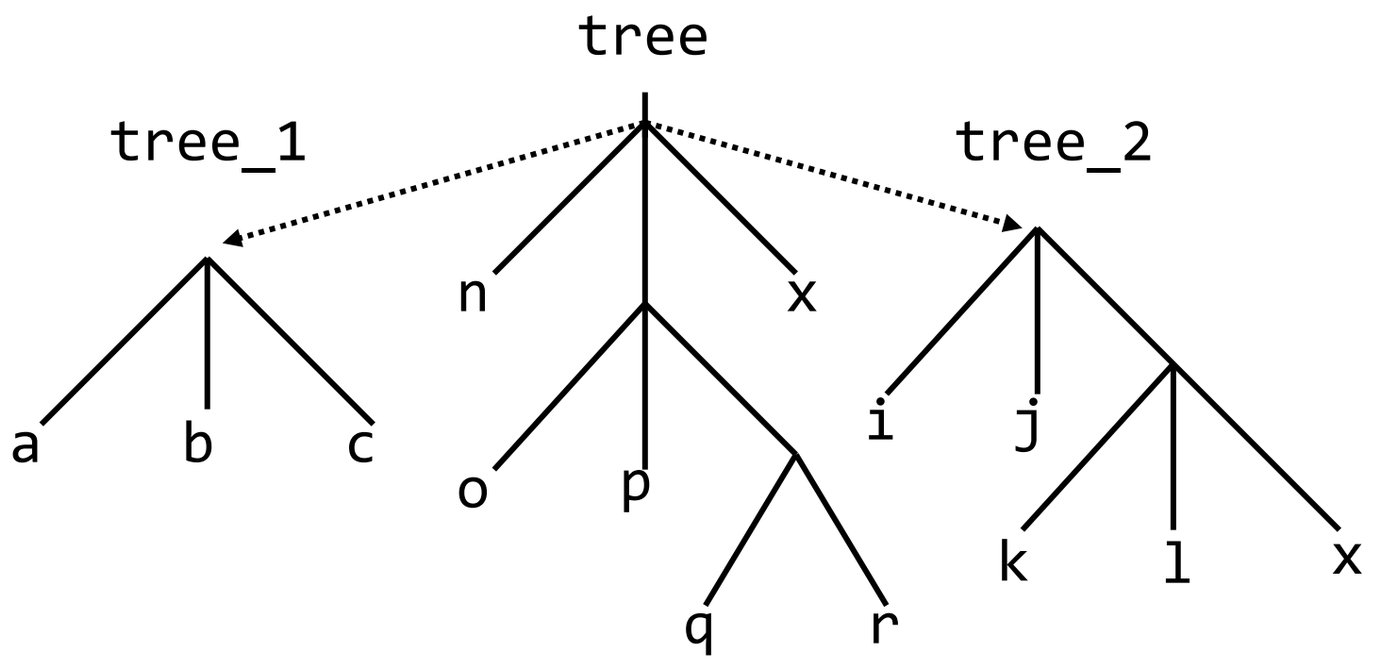
  - bind Tree branches to `TTreeReaderValue` objects
    - type safety by using templated objects
    - possible only with Cling, when JIT is available

Tree Friends

- 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 holding the new tree
  - Create a new Tree holding the branches for the user data
  - Fill the tree/branches with user data
  - Add this new file/tree as friend of the original tree

Tree Friends

- Using Tree Friends




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

Splitting

- Creates one branch per member – recursively

Splitting

- Creates one branch per member – recursively
- Allows to browse objects that are stored in trees, even without their library

Splitting

- Creates one branch per member – recursively
- Allows to browse objects that are stored in trees, even without their library
- Fine grained branches allow fine-grained I/O - read only members that are needed

Splitting

- Creates one branch per member – recursively
- Allows to browse objects that are stored in trees, even without their library
- Fine grained branches allow fine-grained I/O - read only members that are needed
- Supports STL containers too, even vector<T*>

Splitting

- Creates one branch per member – recursively
- Allows to browse objects that are stored in trees, even without their library
- Fine grained branches allow fine-grained I/O - read only members that are needed
- Supports STL containers too, even vector<T*>
- Split level os defined when creating the branches
  - level = 0 no splitting at all
  - level = 1 split only the first-level objects
  - level = 2 split first and second level objects
  - default level is 99 : maximum splitting

Performance Considerations

A split branch is:

Performance Considerations

A split branch is:

- Faster to read – if you only want a subset of data members

Performance Considerations

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

Performance Considerations

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

Performance Considerations

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

- For reading a subset of data recommend to use
  - `branch->GetEntry(ientry)`
  - will read only the required branch data (big difference in case of trees with many branches)

Performance Considerations

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

- For reading a subset of data recommend to use
  - `branch->GetEntry(ientry)`
  - will read only the required branch data (big difference in case of trees with many branches)
- Alternatively can use also
  - `tree->SetBranchStatus("*", 0);`
  - `tree->SetBranchStatus(“myBranch”,1);`

Analyzing Trees: Summary

- Tree is an efficient storage and access for huge amounts of structured data
- Allows selective access of data
- It is used to analyze and select data.
- Most convenient way to analyze data store in a Tree is with the **`TSelector`** class
  - the user creates a new class `MySelector` deriving from **`TSelector`**
  - the `MySelector` object is used in `TTree::Process(TSelector*,...)`
  - ROOT invokes the `TSelector's` functions which are virtuals, so the user provided function implemented in `MySelector` will be called.

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions and solve the exercises on creating, reading and analyzing the Tree

ExerciseTwiki Page

Summary

- The ROOT Tree is one of the most powerful collections available for HEP
- Extremely efficient for huge number of data sets with identical layout
- Very easy to look at `TTree` - use `TBrowser`!
- Write once, read many: ideal for experiments' data; use friends to extend
- Branches allow granular access; use splitting to create branch for each member, even through collections
- `TSelector` class provides a powerful way of processing the Tree data using compiled code

Interactive Data Analysis with PROOF

- Bleeding Edge Physics
  with
  Bleeding Edge Computing

Dr Lorenzo Moneta
CERN PH-SFT
CH-1211 Geneva 23
sftweb.cern.ch
root.cern.ch

PROOF

PROOF

Huge amounts of events, hundreds of CPUs

PROOF

Huge amounts of events, hundreds of CPUs

Split the job into N events / CPU!

PROOF

Huge amounts of events, hundreds of CPUs

Split the job into N events / CPU!

PROOF for TSelector based analysis:

PROOF

Huge amounts of events, hundreds of CPUs

Split the job into N events / CPU!

PROOF for TSelector based analysis:

- start analysis locally ("client"),

PROOF

Huge amounts of events, hundreds of CPUs

Split the job into N events / CPU!

PROOF for TSelector based analysis:

- start analysis locally ("client"),
- PROOF distributes data and code,

PROOF

Huge amounts of events, hundreds of CPUs

Split the job into N events / CPU!

PROOF for TSelector based analysis:

- start analysis locally ("client"),
- PROOF distributes data and code,
- lets CPUs ("workers") run the analysis,

PROOF

Huge amounts of events, hundreds of CPUs

Split the job into N events / CPU!

PROOF for TSelector based analysis:

- start analysis locally ("client"),
- PROOF distributes data and code,
- lets CPUs ("workers") run the analysis,
- collects and combines (merges) data,

PROOF

Huge amounts of events, hundreds of CPUs

Split the job into N events / CPU!

PROOF for TSelector based analysis:

- start analysis locally ("client"),
- PROOF distributes data and code,
- lets CPUs ("workers") run the analysis,
- collects and combines (merges) data,
- shows analysis results locally

Interactive!

- Start analysis

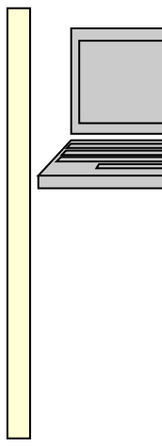
Interactive!

- Start analysis
- Watch status while running

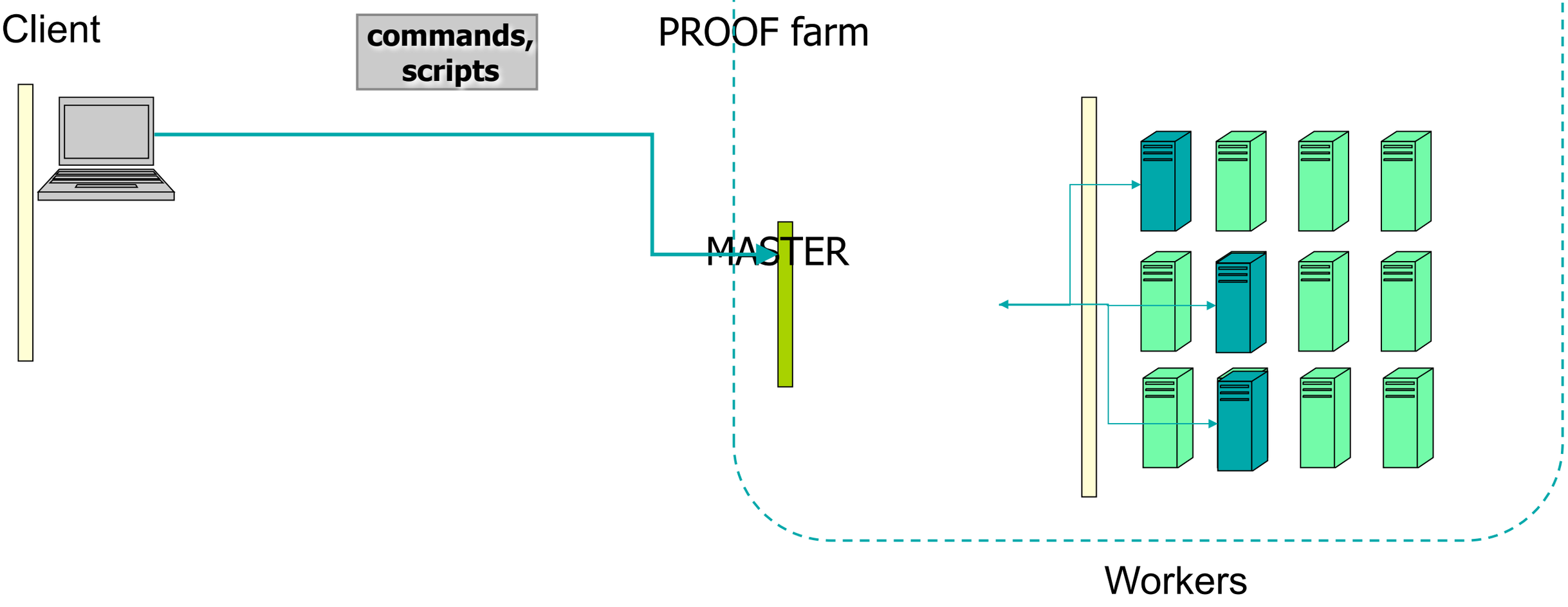
Interactive!

- Start analysis
- Watch status while running
- Forgot to create a histogram?
  - Interrupt the process
  - Modify the selector
  - Re-start the analysis
- More dynamic than a batch system

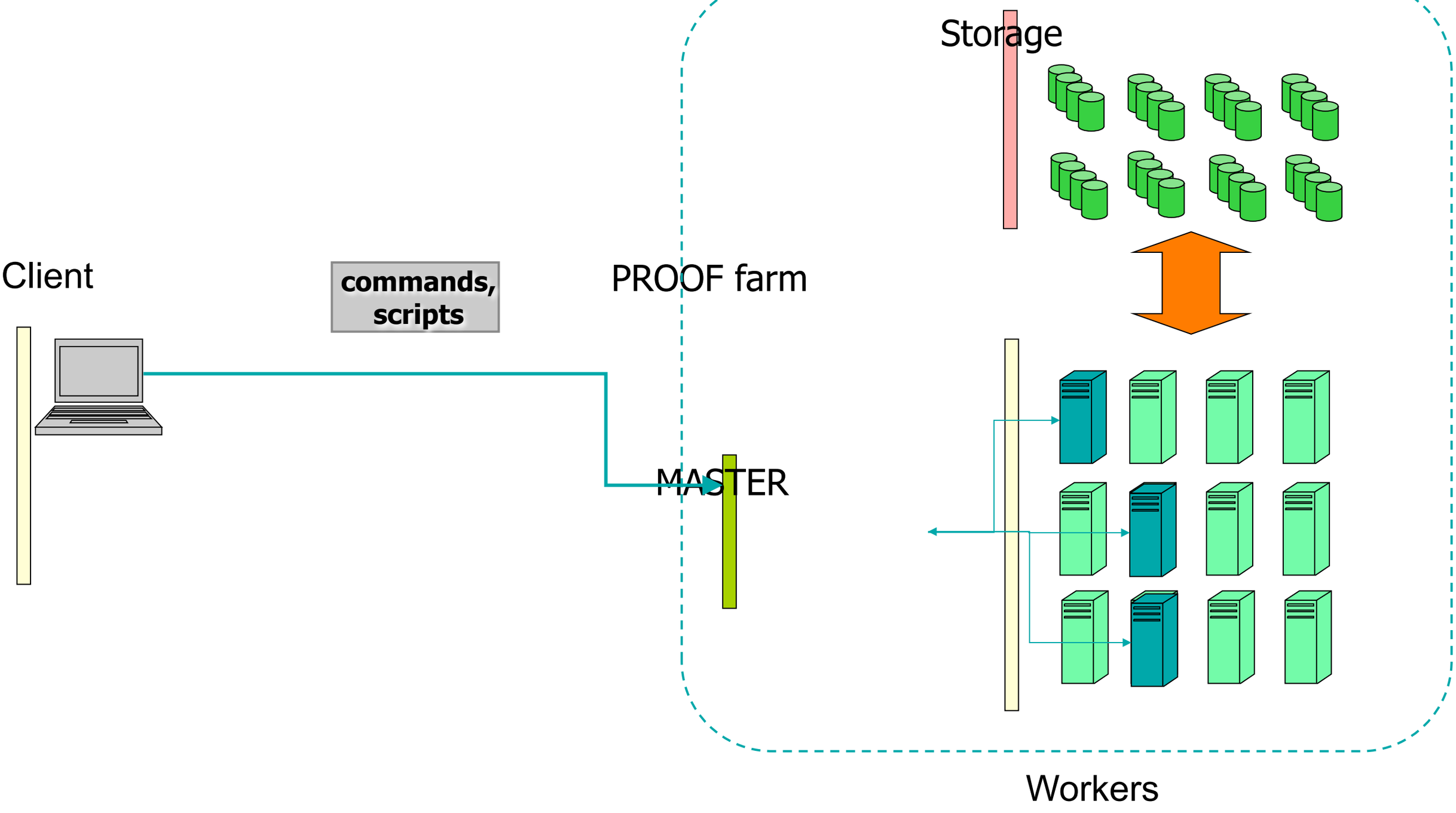
PROOF



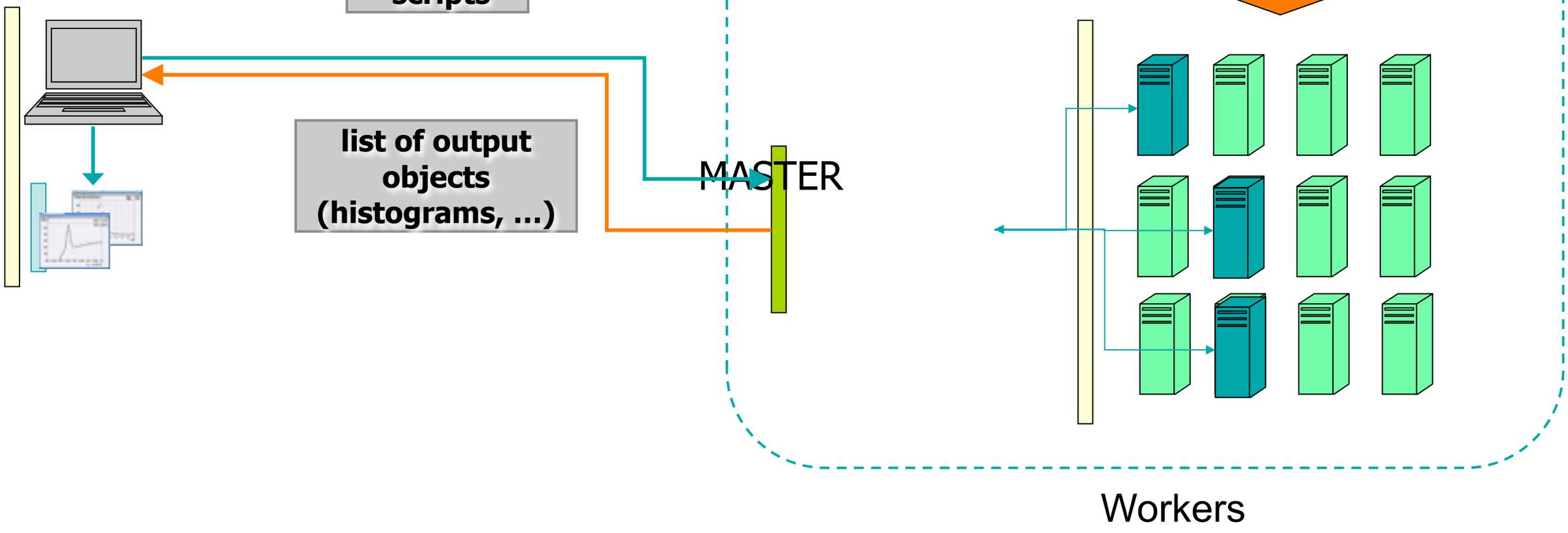
PROOF



PROOF


Client
commands, scripts
PROOF farm
Storage
MASTER
Workers

PROOF


Client
commands, scripts
PROOF farm
Storage
MASTER
list of output objects (histograms, ...)
Workers

Scheduling

- Decides where to run which (part of) the jobs
- E.g. simple batch system
- Can autonomously split jobs into parts (“packets”)
- Involves
  - resource management (CPU, I/O, memory)
  - data locality
  - priorities (jobs / users)
  - and whatever other criteria are deemed relevant
- Often optimizing jobs’ distribution towards overall goal: maximum CPU utilization (Grid), minimum time to result (PROOF)

Packetizer Role and Goals

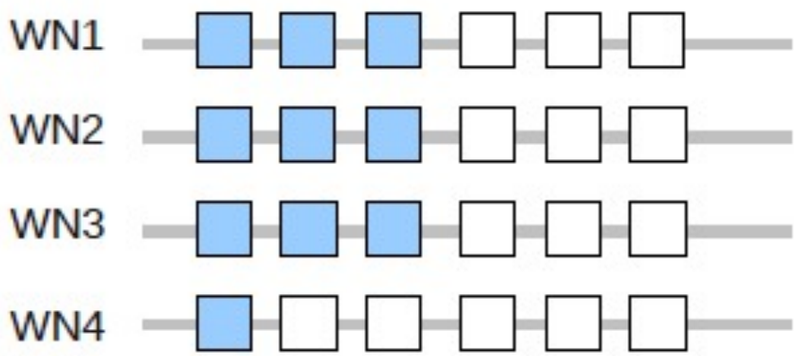
- Distributes units of work ("packets") to workers
- Grid's packet: >=1 file
- Result arrives when last resource has processed last file:

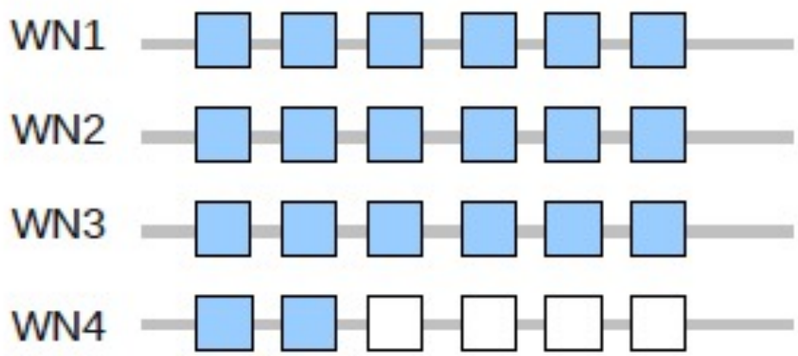
$$t = t_{init} + \max_{jobs}(R_i \cdot N_i^{files}) + t_{final}$$

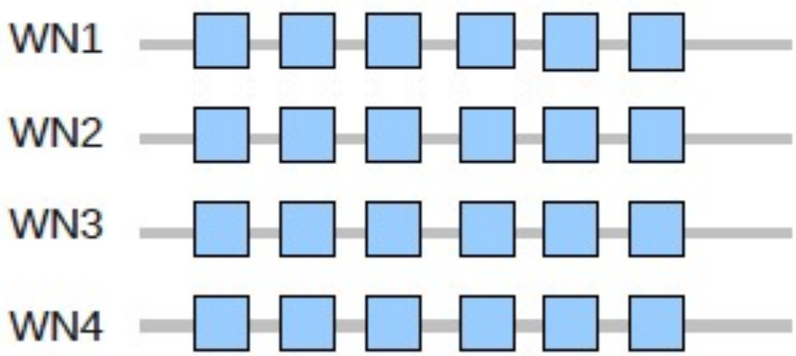
  - t_{init}, t_{final}: time to initialize / finalize the jobs
    R_i: processing rate of job i
    N_i^{files}: number of files for job i
- Result:
  - slowest job defines running time
  - large tail in CPU utilization versus time

Static…

- Example: 24 files on 4 worker nodes, one under-performing







The slowest worker node sets the processing time

PROOF's Dynamic Packetizer

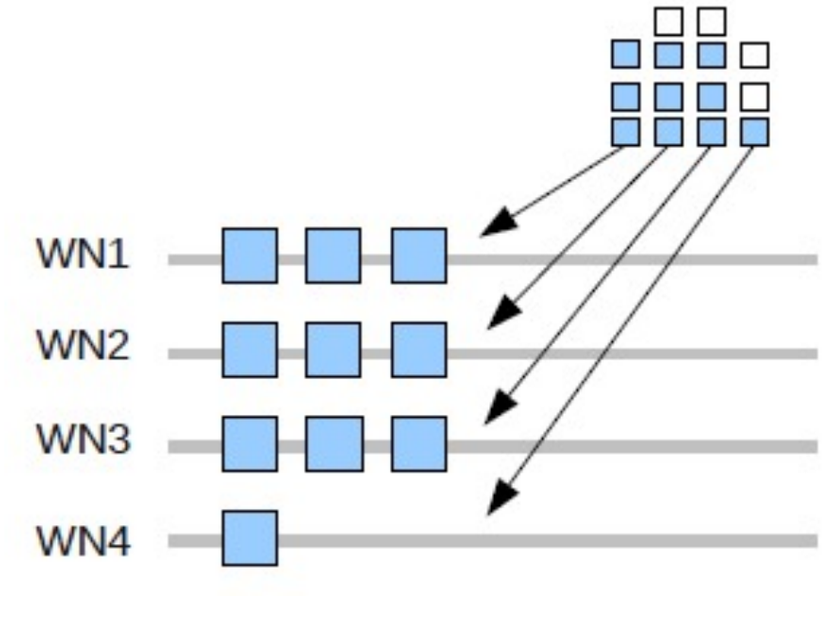
- PROOF packetizer's goal: results as early as possible
- All workers should finish at the same time:

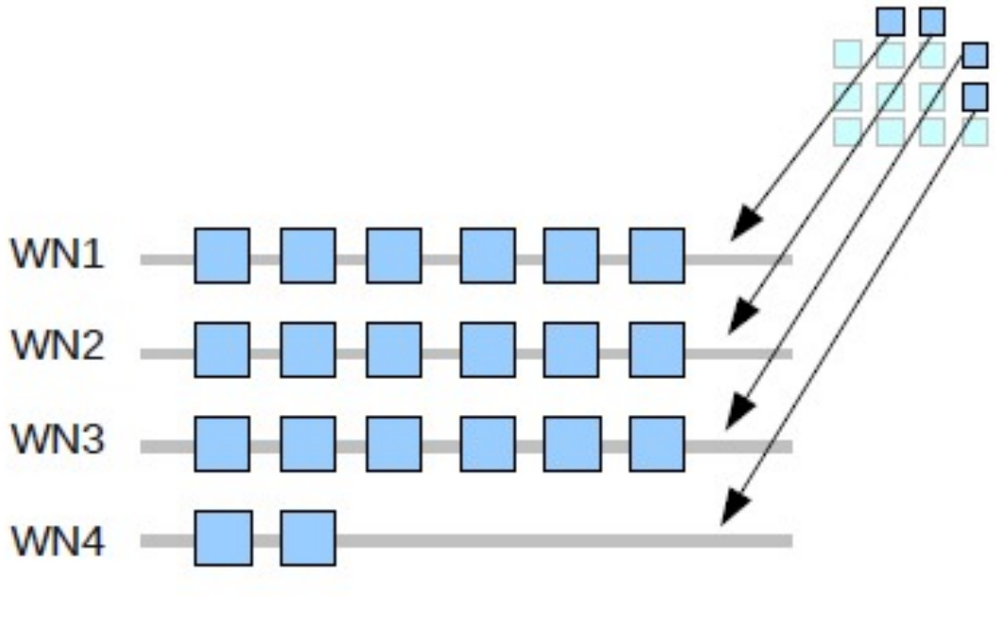
$$t = t_{init} + \max_{jobs}(R_i \cdot N_i^{files}) + t_{final}$$

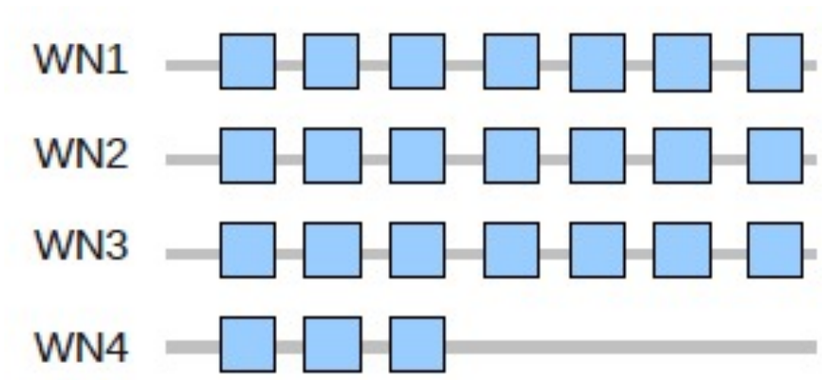
ideally, $R_i \cdot N_i^{files}$ equal for all jobs

- Cannot reliably predict performance (R_i) of workers
  - job interaction, e.g. number of jobs accessing the same disk
  - CPU versus I/O duty of jobs
- Instead: update prediction based on real-time past performance while running
- Pull architecture: workers ask for new packets

Dynamic Packet Distribution







The slowest worker node gets less work to do: the processing time is less affected by its under performance

Creating a session

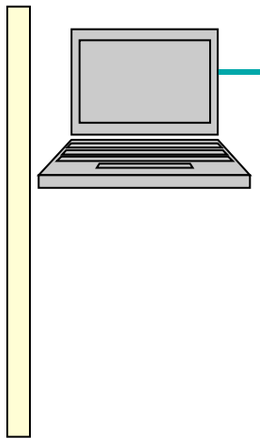
To create a PROOF session from the ROOT prompt, just type:

```
TProof::Open("master")
```

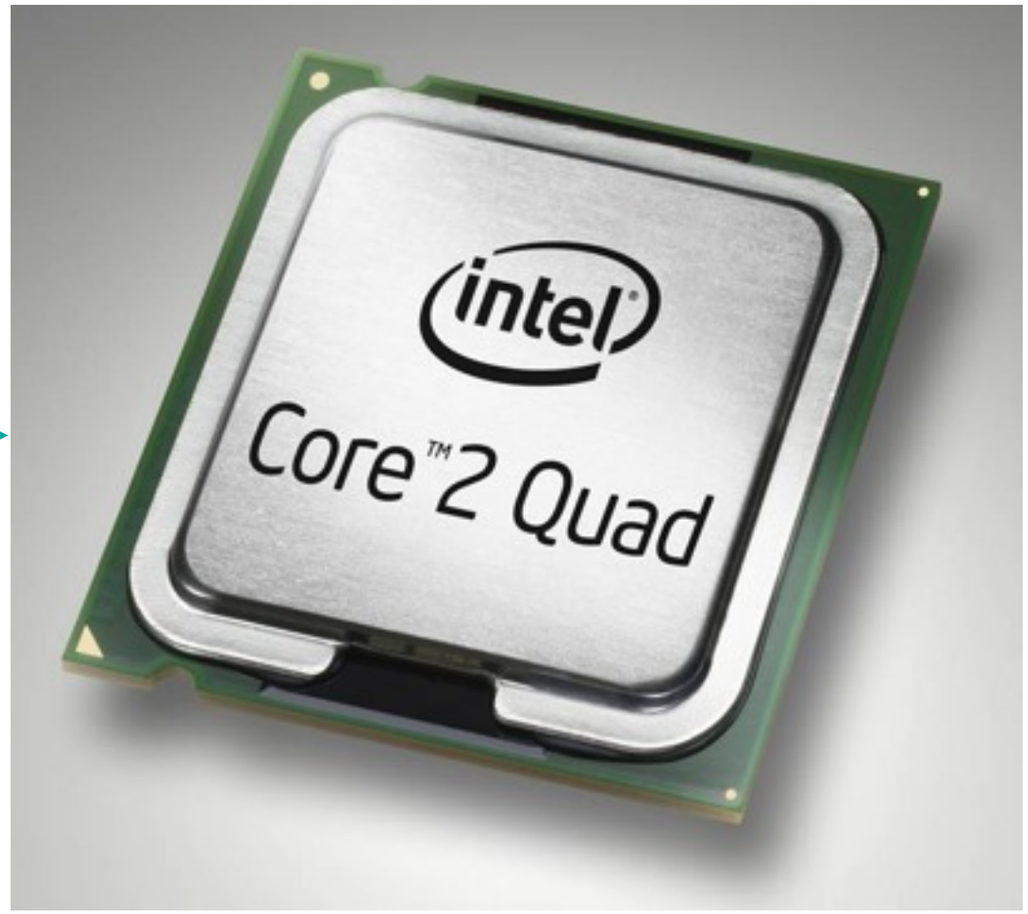
where "master" is the hostname of the master machine on the PROOF cluster

PROOF Lite

Client

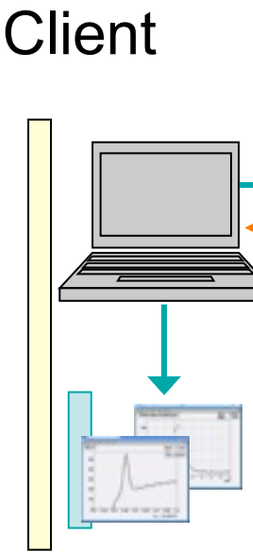


commands, scripts



Multi-core Desktop/Laptop

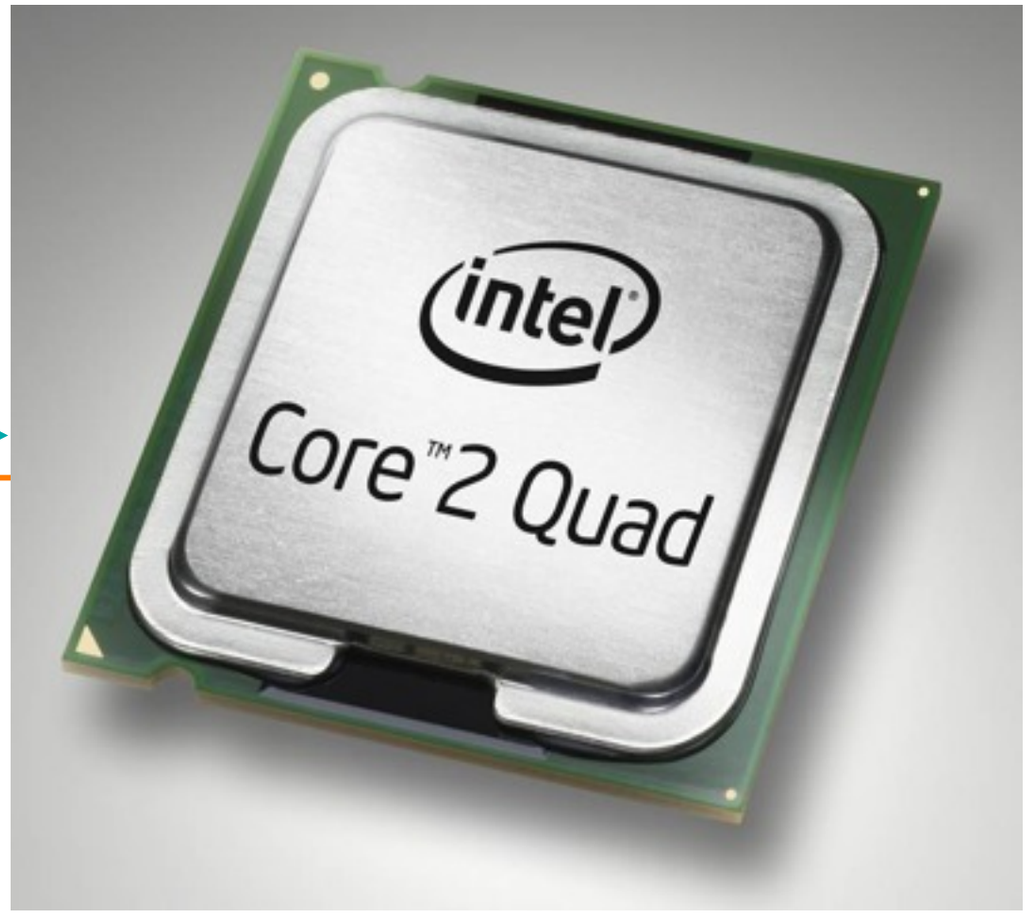
PROOF Lite



Client

commands, scripts

list of output objects (histograms, …)



Multi-core Desktop/Laptop

What is PROOF Lite?

- PROOF optimized for single many-core machines
- Zero configuration setup
  - No config files and no daemons
- Like PROOF it can exploit fast disks, SSD's, lots of RAM, fast networks and fast CPU's
- If your code works on PROOF, then it works on PROOF Lite and vice versa

Creating a session

To create a PROOF Lite session from the ROOT prompt, just type:

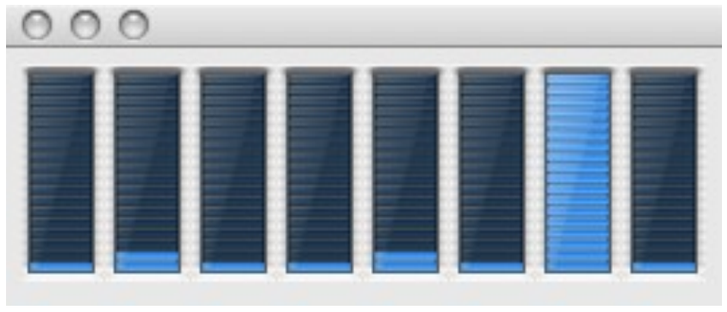
```
TProof::Open("")
```

Then you can use your multicore computer as a PROOF cluster!

PROOF Analysis

- Example of local TChain analysis

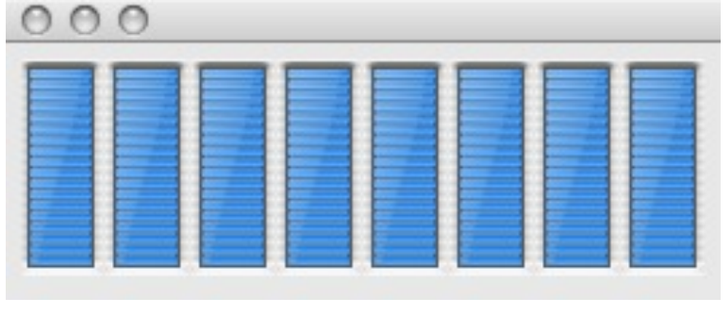
```
// Create a chain of trees
root[0] TChain *c = new TChain("myTree");
root[1] c->Add("http://www.any.where/file1.root");
root[2] c->Add("http://www.any.where/file2.root");

// MySelector is a TSelector
root[3] c->Process("MySelector.C+");
```



PROOF Analysis

- Same example with PROOF

```
// Create a chain of trees
root[0] TChain *c = new TChain("myTree");
root[1] c->Add("http://www.any.where/file1.root");
root[2] c->Add("http://www.any.where/file2.root");

// Start PROOF and tell the chain to use it
root[3] TProof::Open("");
root[4] c->SetProof();

// Process goes via PROOF
root[5] c->Process("MySelector.C+");
```



TSelector & PROOF

TSelector & PROOF

- Begin() called on the client only

TSelector & PROOF

- Begin() called on the client only
- SlaveBegin() called on each worker: create histograms

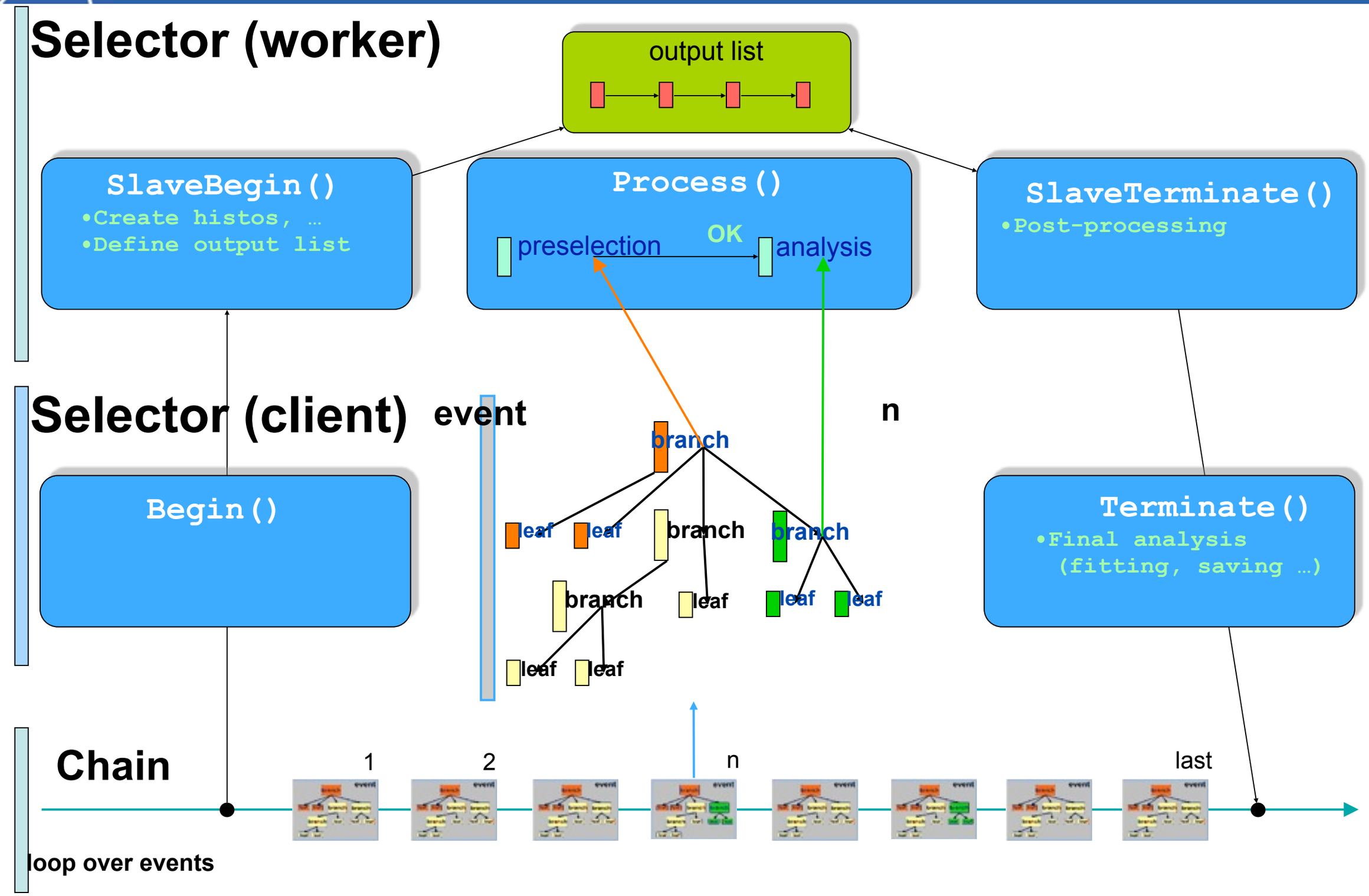
TSelector & PROOF

- Begin() called on the client only
- SlaveBegin() called on each worker: create histograms
- SlaveTerminate() rarely used; post processing of partial results before they are sent to master and merged

TSelector & PROOF

- Begin() called on the client only
- SlaveBegin() called on each worker: create histograms
- SlaveTerminate() rarely used; post processing of partial results before they are sent to master and merged
- Terminate() runs on the client: save results, display histograms, …

PROOF Analysis



Output List (result of the query)

- Each worker has a partial output list
- Objects have to be added to the list in TSelector::SlaveBegin() e.g.:

```
fHist = new TH1F("h1", "h1", 100, -3., 3.);
fOutput->Add(fHist);
```

- At the end of processing the output list gets sent to the master
- The Master merges objects and returns them to the client. Merging is e.g. "Add()" for histograms, appending for lists and trees

Example

```
void MySelector::SlaveBegin(TTree *tree) {
   // create histogram and add it to the output list
   fHist = new TH1F("MyHist","MyHist",40,0.13,0.17);
   GetOutputList()->Add(fHist);
}

Bool_t MySelector::Process(Long64_t entry) {
   my_branch->GetEntry(entry); // read branch
   fHist->Fill(my_data);       // fill the histogram
   return kTRUE;
}

void MySelector::Terminate() {
   fHist->Draw();              // display histogram
}
```

Results

At the end of Process(), the output list is accessible via
gProof->GetOutputList()

Results

At the end of Process(), the output list is accessible via gProof->GetOutputList()

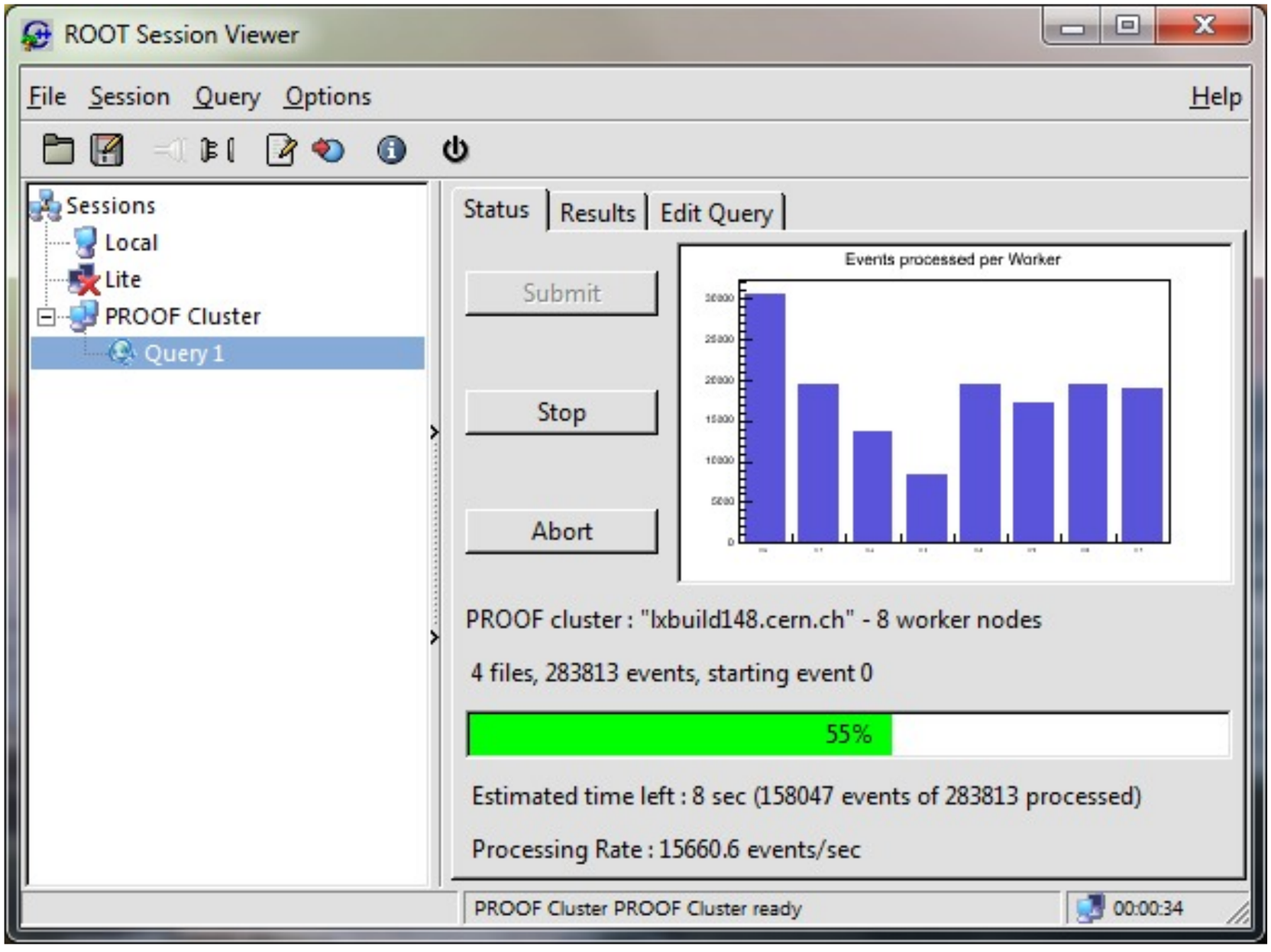
```
// Get the output list
root[0] TList *output = gProof->GetOutputList();

// Retrieve 2D histogram "h2"
root[1] TH2F *h2 = (TH2F*)output->FindObject("h2");

// Display the histogram
root[2] h2->Draw();
```

PROOF GUI Session

Starting a PROOF GUI session is trivial:

```
TProof::Open()
```

Opens GUI:



PROOF Documentation

Documentation available online at

http://root.cern.ch/drupal/content/proof

But of course you need a little cluster of CPUs

Like your multi-core game console!

Summary

- We are at the end
  - still to complete the exercise on PROOF
- We have learned about ROOT
  - general overview of ROOT
  - some basics of ROOT I/O
  - tree and their use for data analysis
  - PROOF for parallel analysis of Trees

- Not cover other issues like fitting and RooFit/ RooStats or TMVA for statistical data analysis
  - see for example one of the past statistical school at Desy

Summary

Looking forward to hearing from you:

- as a user (help! bug! suggestion!)
- and as a developer! (with new code contributions)

Watch out for the new production version of ROOT 6 with the new Cling interpreter

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions and solve the exercises on data analysis using PROOF

ExerciseTwiki Page