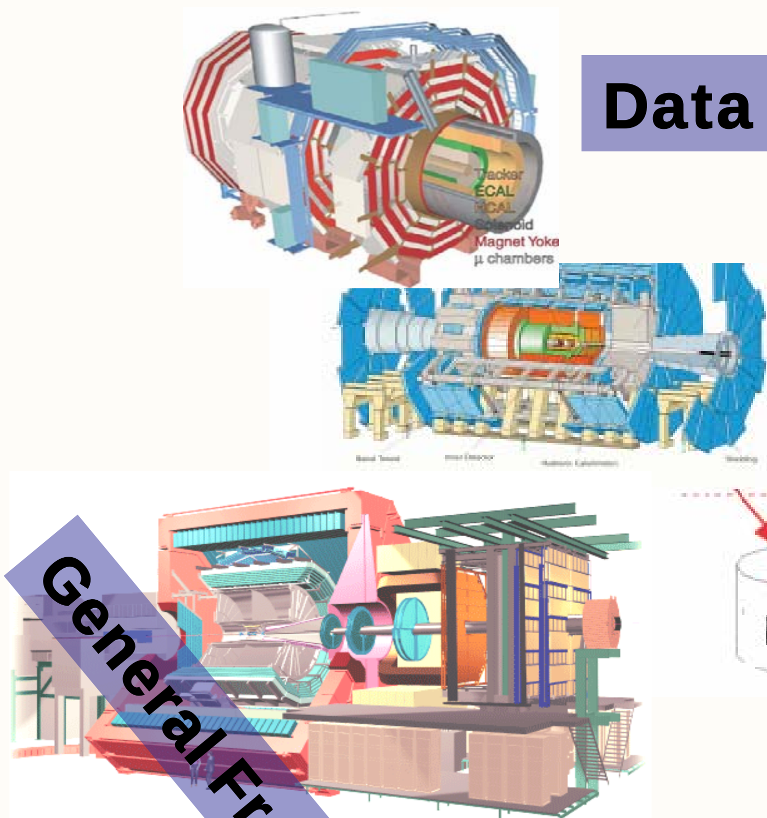
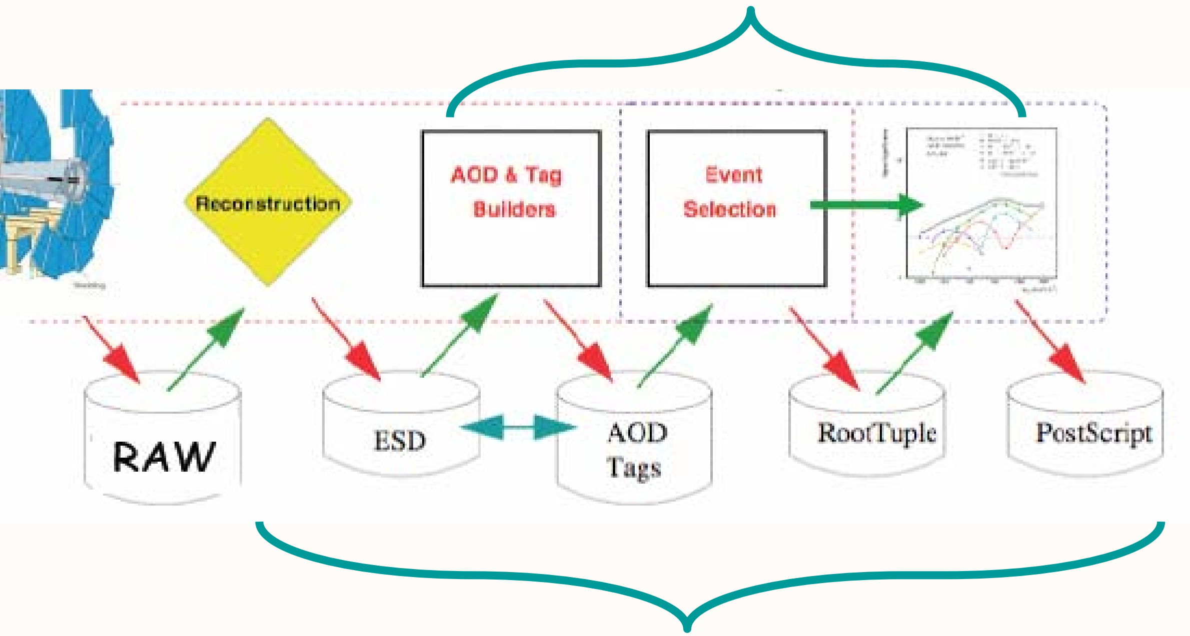
ROOT

Bertrand Bellenot, Axel Naumann
CERN

ROOT Application Domains

Data Analysis & Visualization





General Framework

Data Storage: Local, Network

ROOT in a Nutshell

Object Oriented framework for large scale data handling applications, written in C++

Provides, among others,

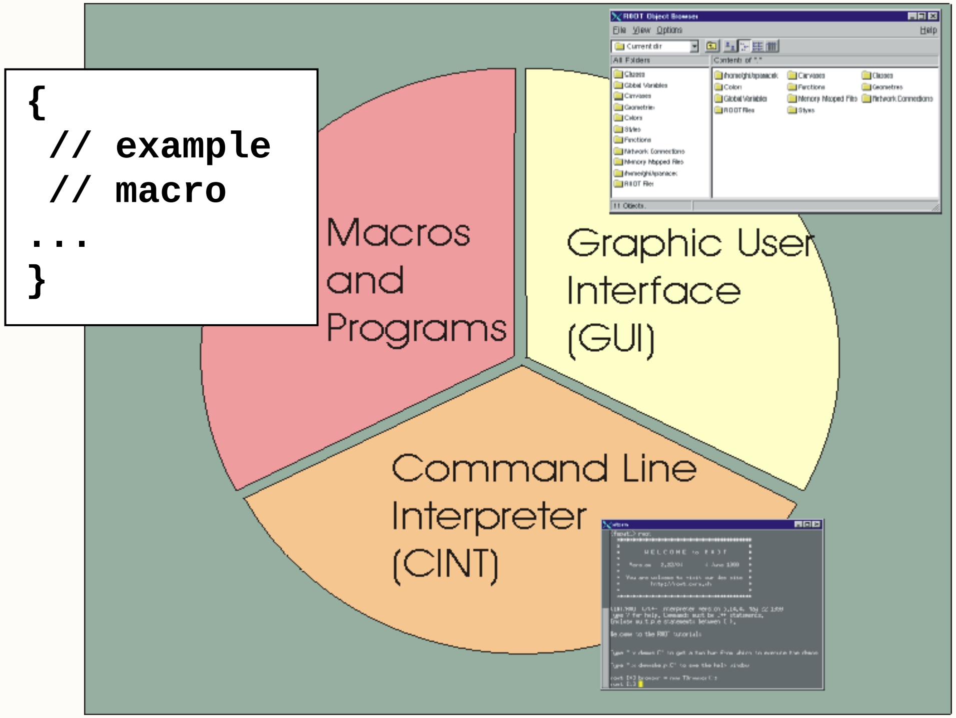
- an efficient data storage, access and query system (PetaBytes)
- a C++ interpreter
- advanced statistical analysis algorithms (multi dimensional histogramming, fitting, minimization and cluster finding)
- scientific visualization: 2D and 3D graphics, Postscript, PDF, LateX
- geometrical modeller
- PROOF parallel query engine

ROOT Library Structure

ROOT libraries are a layered structure

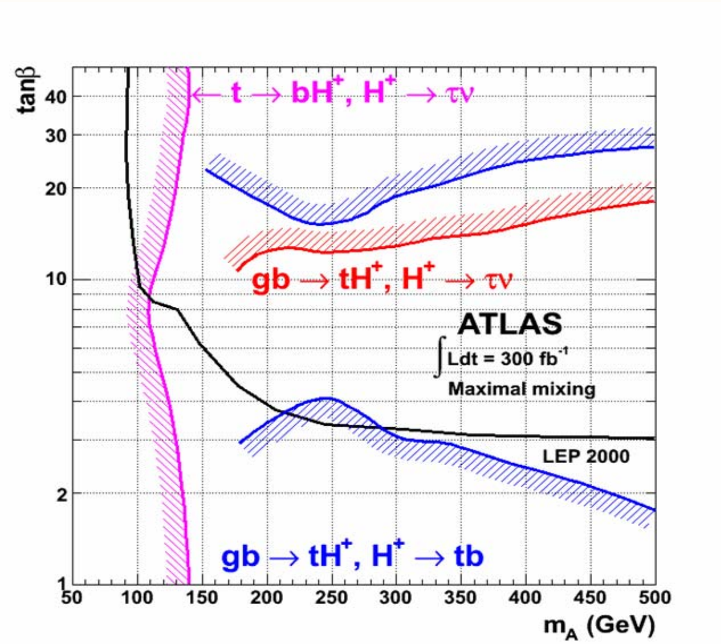
- CORE classes always required: support for RTTI, basic I/O and interpreter
- Optional libraries loaded when needed. Separation between data objects and the high level classes acting on these objects. Example: a batch job uses only the histogram library, no need to link histogram painter library
- Shared libraries reduce the application size and link time
- Mix and match, build your own application
- Reduce dependencies via plug-in mechanism

Three User Interfaces

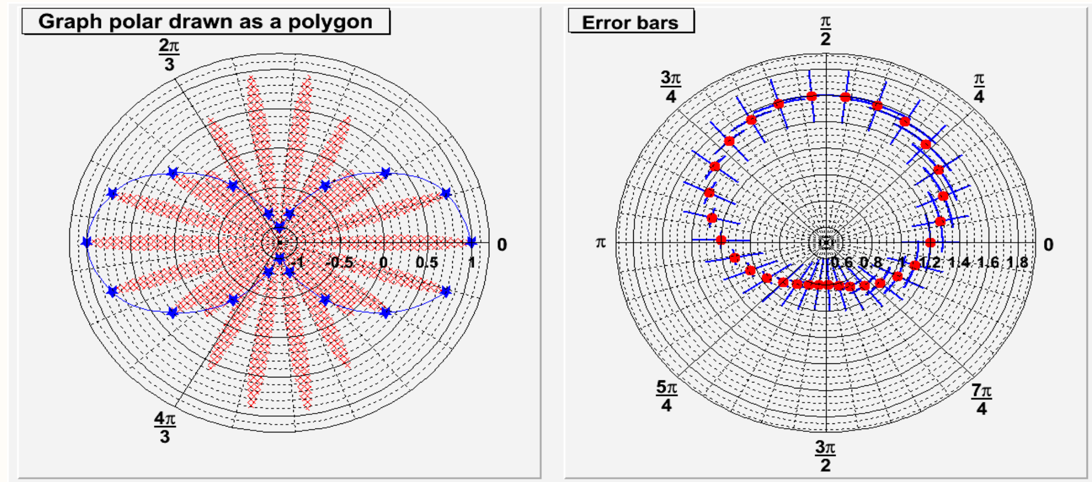


- GUI
  windows, buttons, menus
- Command line
  CINT (C++ interpreter), Python, Ruby,…
- Macros, applications, libraries (C++ compiler and interpreter)

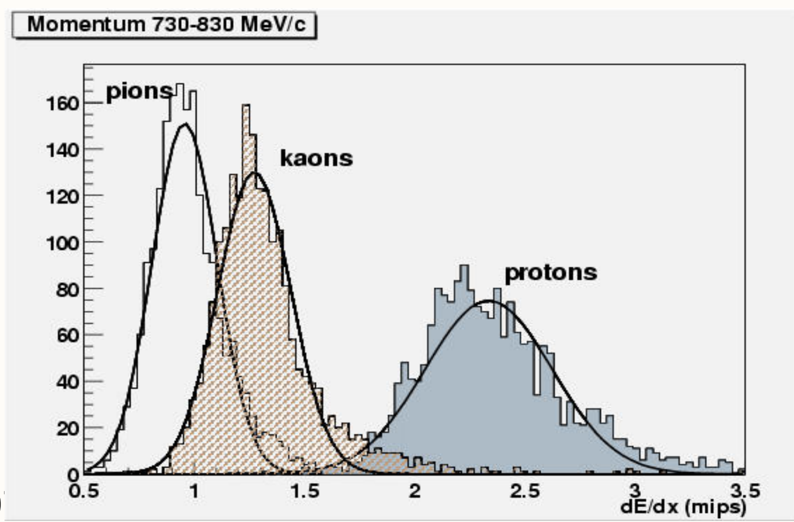
Graphics



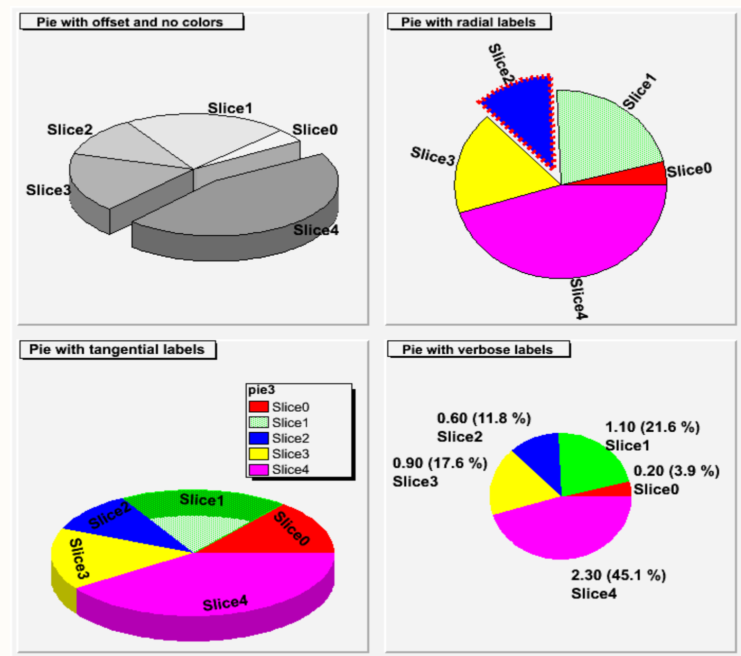
tanβ
t → bH⁺, H⁺ → τν
gb → tH⁺, H⁺ → τν
ATLAS
∫Ldt = 300 fb⁻¹
Maximal mixing
LEP 2000
gb → tH⁺, H⁺ → tb
m_A (GeV)



Graph polar drawn as a polygon
Error bars

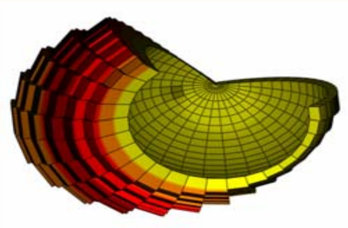


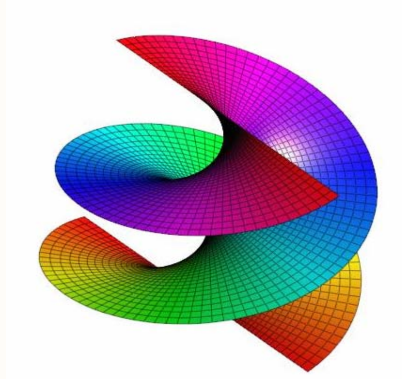
Momentum 730-830 MeV/c
pions
kaons
protons
dE/dx (mips)

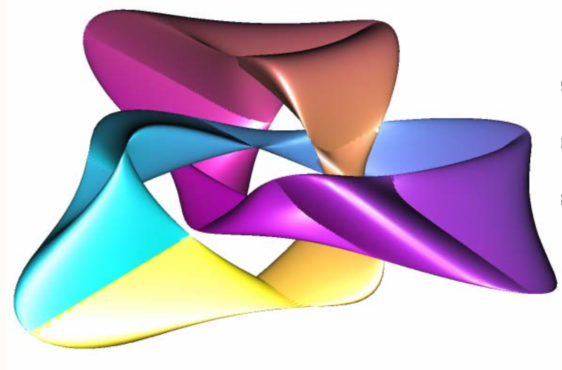


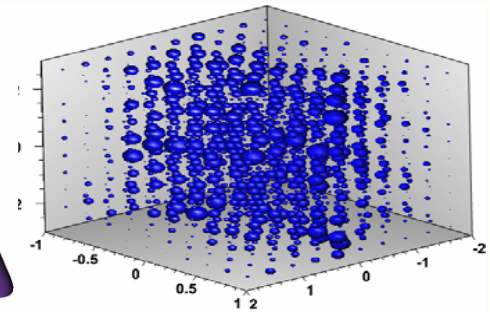
Pie with offset and no colors
Pie with radial labels
Pie with tangential labels
Pie with verbose labels
pie3
Slice0
Slice1
Slice2
Slice3
Slice4
0.60 (11.8 %)
1.10 (21.6 %)
0.90 (17.6 %)
0.20 (3.9 %)
2.30 (45.1 %)

Graphics (2D-3D)

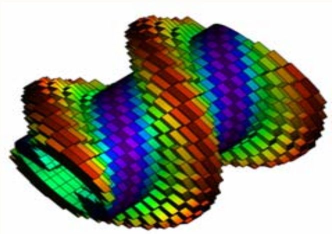


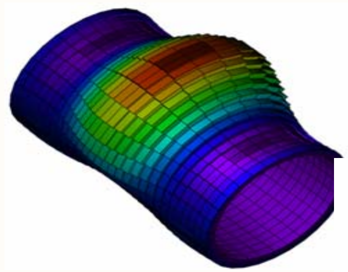




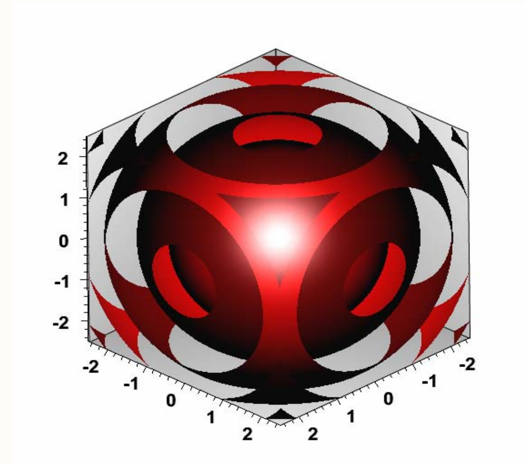


TH3

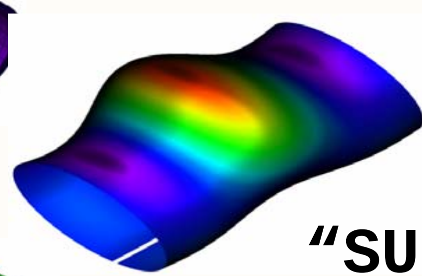




TGLParametric

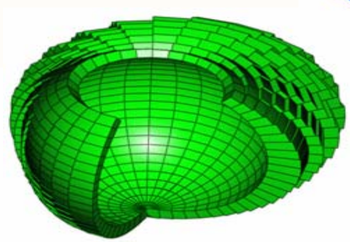


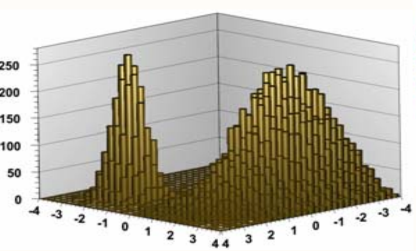
“LEGO”

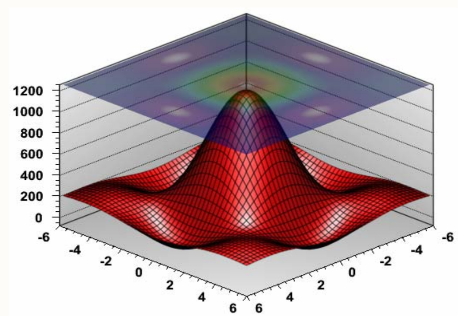


“SURF”

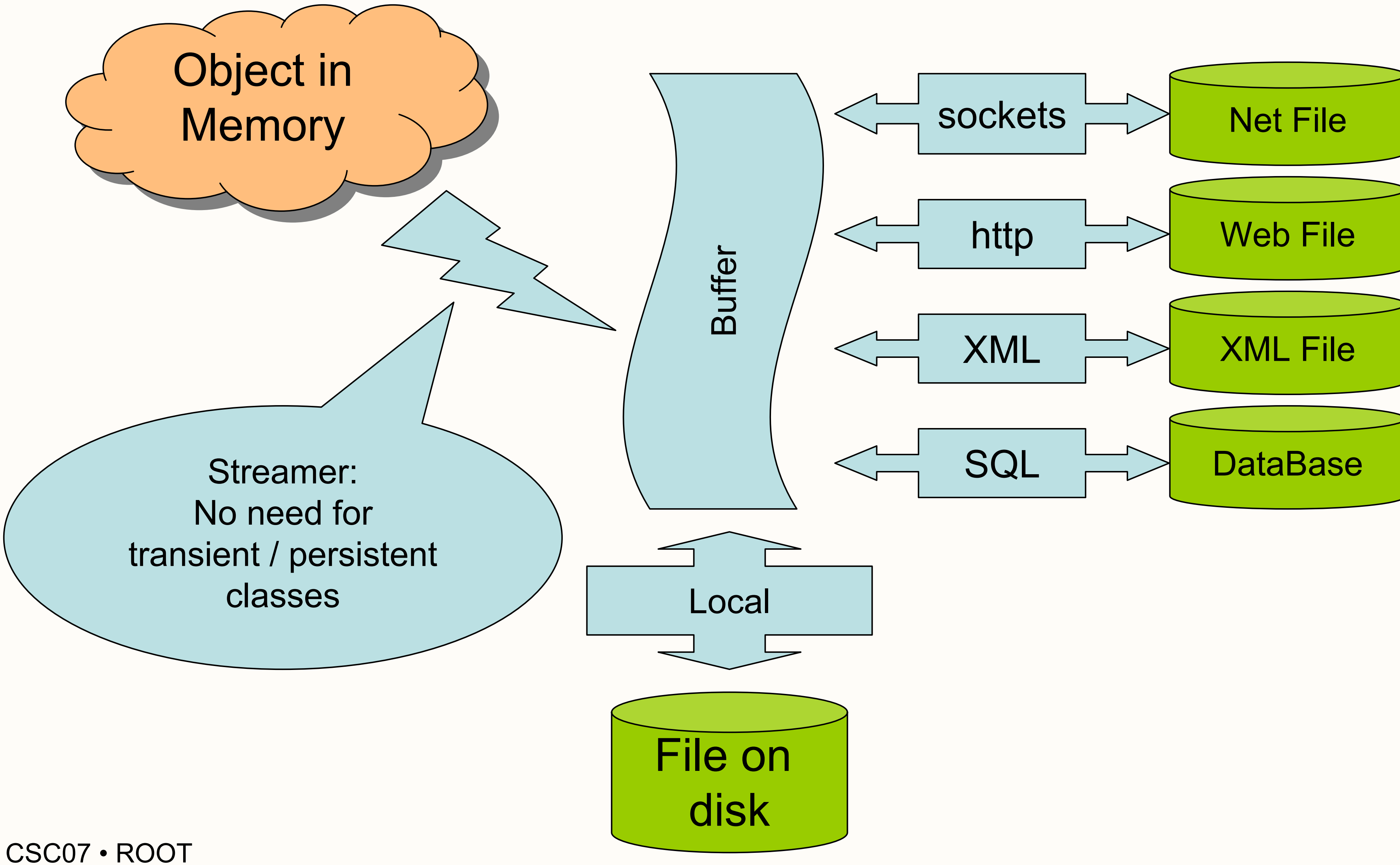
TF3



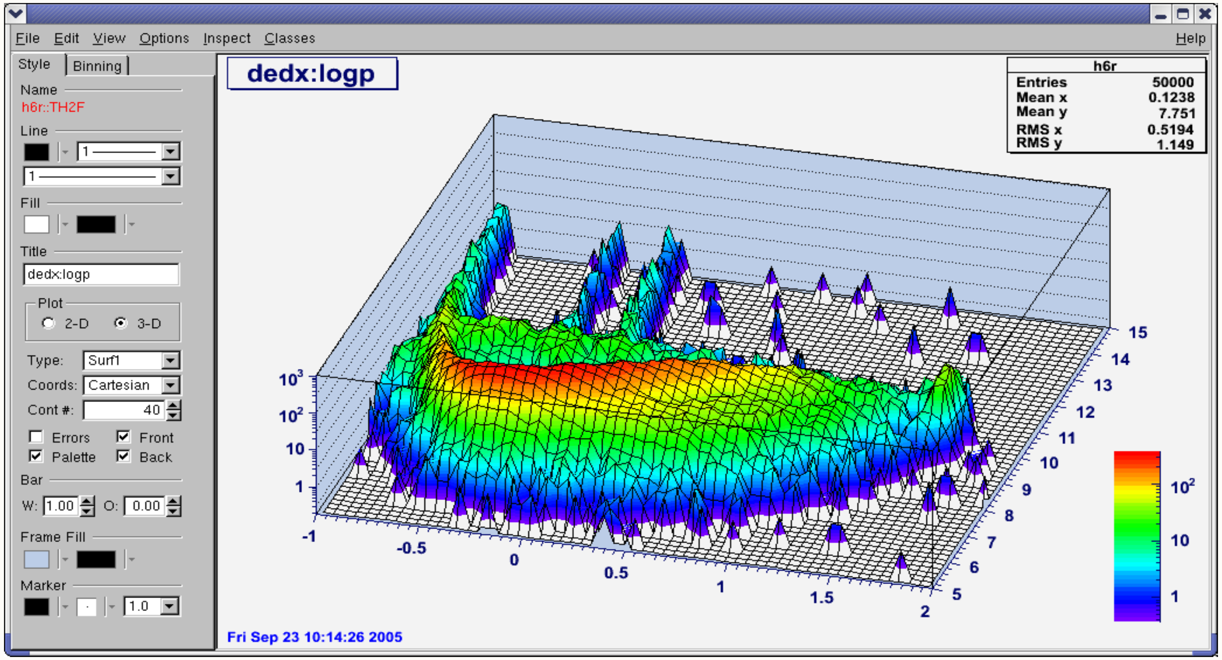


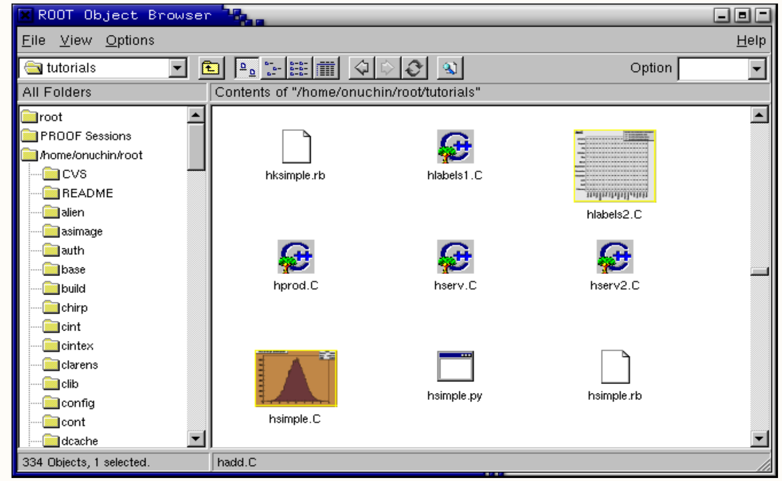


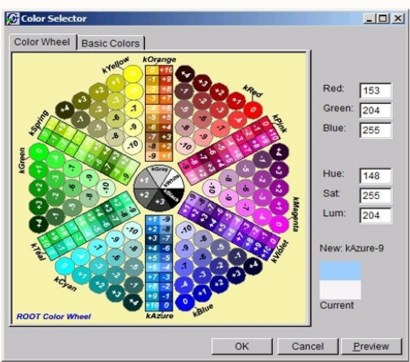
I/O

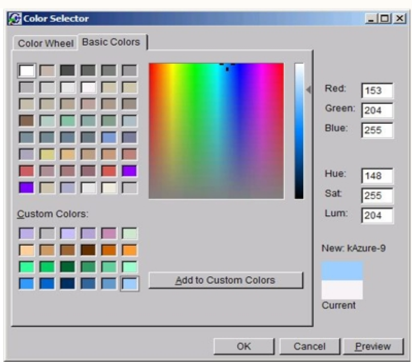

Object in Memory
Streamer: No need for transient / persistent classes
Buffer
sockets
http
XML
SQL
Net File
Web File
XML File
DataBase
Local
File on disk

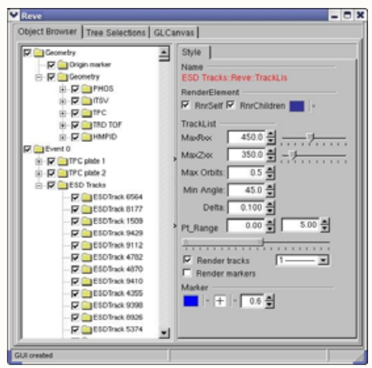
GUI (Graphical User Interface)

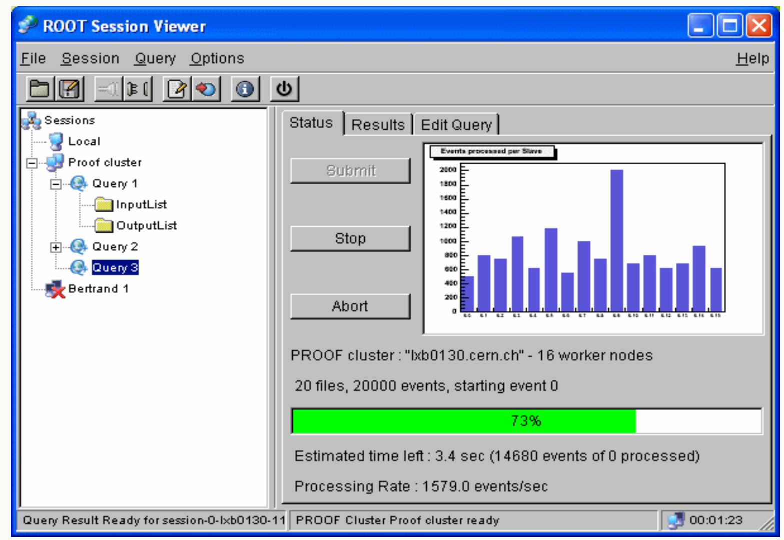




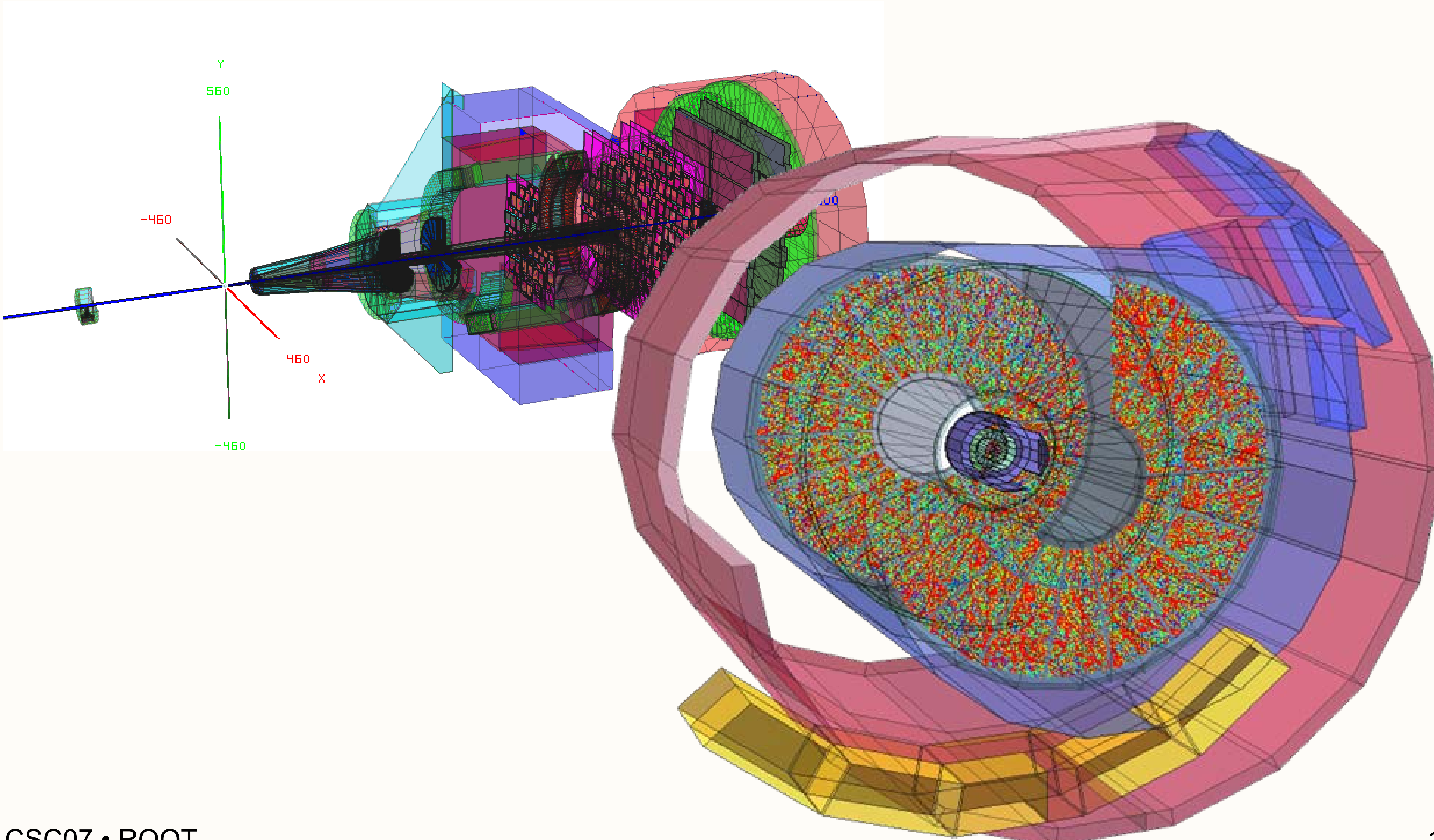




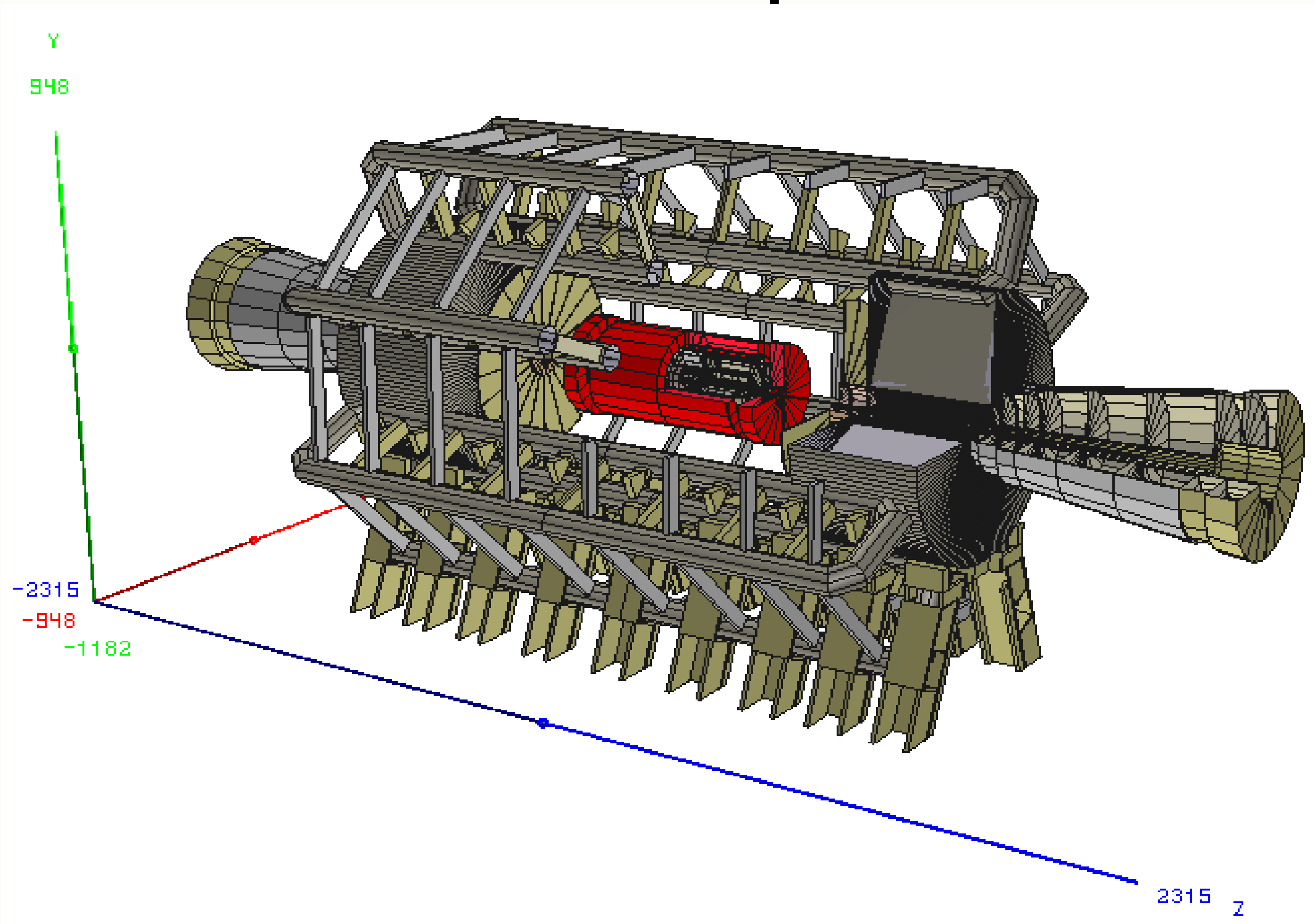




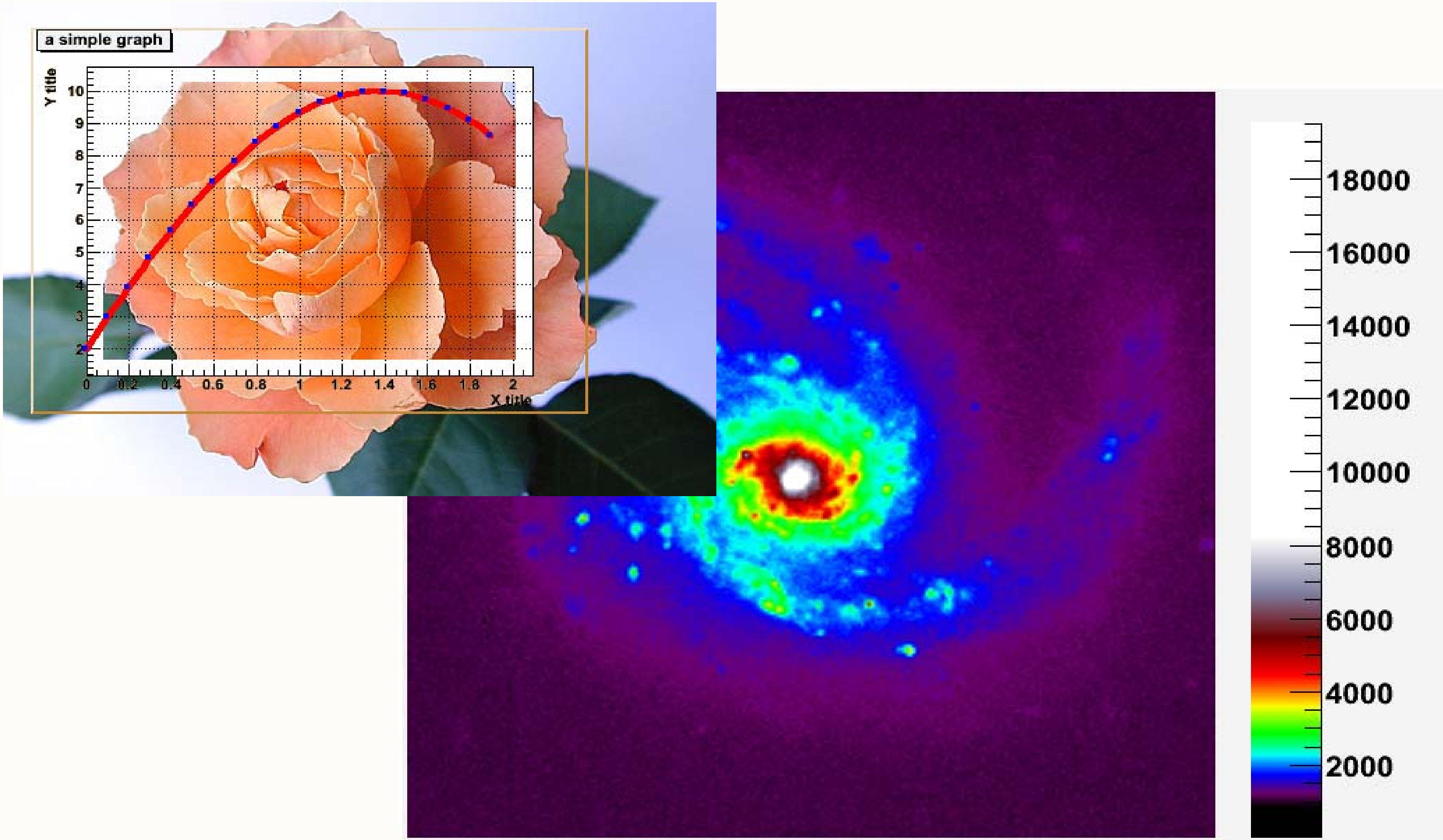
Geometry



OpenGL



ASImage


a simple graph
Y title
X title
0 0.2 0.4 0.6 0.8 1 1.2 1.4 1.6 1.8 2
2 3 4 5 6 7 8 9 10
18000
16000
14000
12000
10000
8000
6000
4000
2000

HTML

class TAttMarker
library: libCore
#include "TAttMarker.h"
Display options:
- [ ] Show inherited
- [x] Show non-public

[↑ Top] | [? Help]

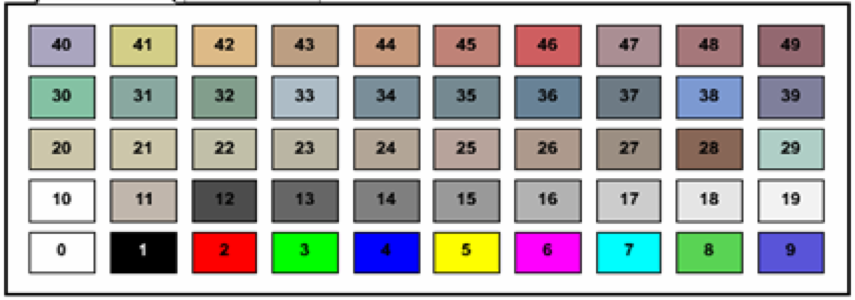
Location:	ROOT » BASE » TAttMarker
Quick Links:	ROOT | Class Index | Class Hierarchy
Source:	header file | source file | viewCVS header | viewCVS source
Sections:	class description | function members | data members | class chart

Int_t **GetQuantiles** (Int_t nprobSum, Double_t* q, con

```
Compute Quantiles for density distribu
   Quantile x_q of a probability distr
```

$F(x_q) = \int_{xmin}^{x_q} f\,dx = q$ with $0 <= q <=$

Picture | Source



Picture | Source

```
{
   TCanvas *c = new TCanvas("c","Fill Area colors",0,0,500,200);
   c.DrawColorTable();
   return c;
}
```

Data Members

protected:

Color_t	**fMarkerColor**	Marker color index
Size_t	**fMarkerSize**	Marker size
Style_t	**fMarkerStyle**	Marker style

```
{
Load = kFALSE;
sEdit::IsSTLCont(name);
ter->CheckClassInfo(name)) shouldLoad = kTRUE;
else if (fImplFileLine>=0) {
   // If the TClass is being generated from a ROOT dictionary,
```

int TClassEdit::IsSTLCont(const char* type,int testAlloc=0)

Math

Histogram library

- TH1
- TF1

MathMore

- Random Numbers
- Extra algorithms
- Extra Math functions
- **GSL and more**

MathCore

- Function interfaces
- Physics Vectors
- Basic algorithms
- Basic Math functions

Statistical Libraries

- Statistical Utilities
- TMVA
- MLP

Fitting and Minimization

- New Fitter
- RooFit
- TMinuit
- TFumili
- Linear Fitter
- Minuit2 (new C++ Minuit)

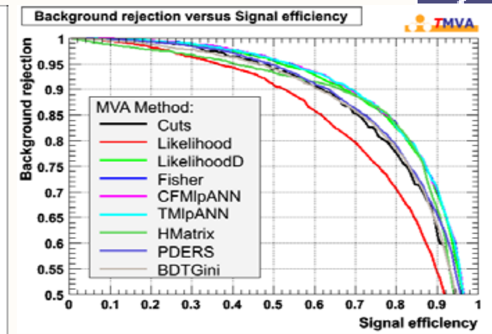
Linear Algebra

- TMatrix
- SMatrix

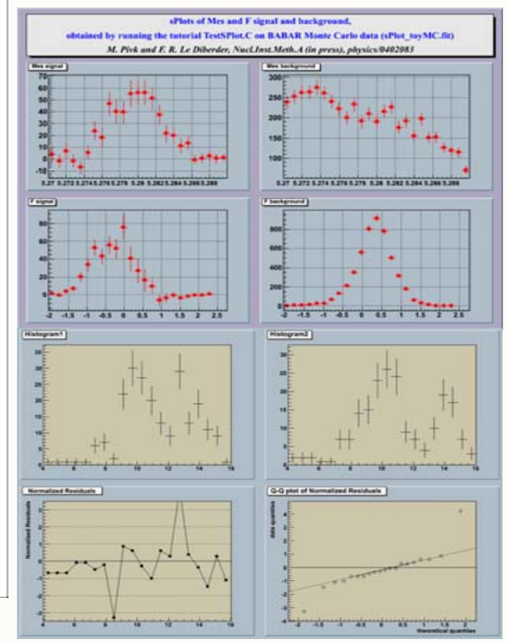
libCore

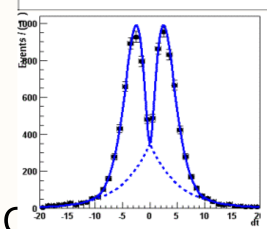
- TMath
- TRandom

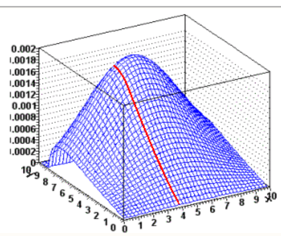
TMVA

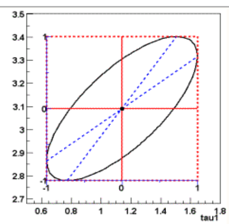


SPlot

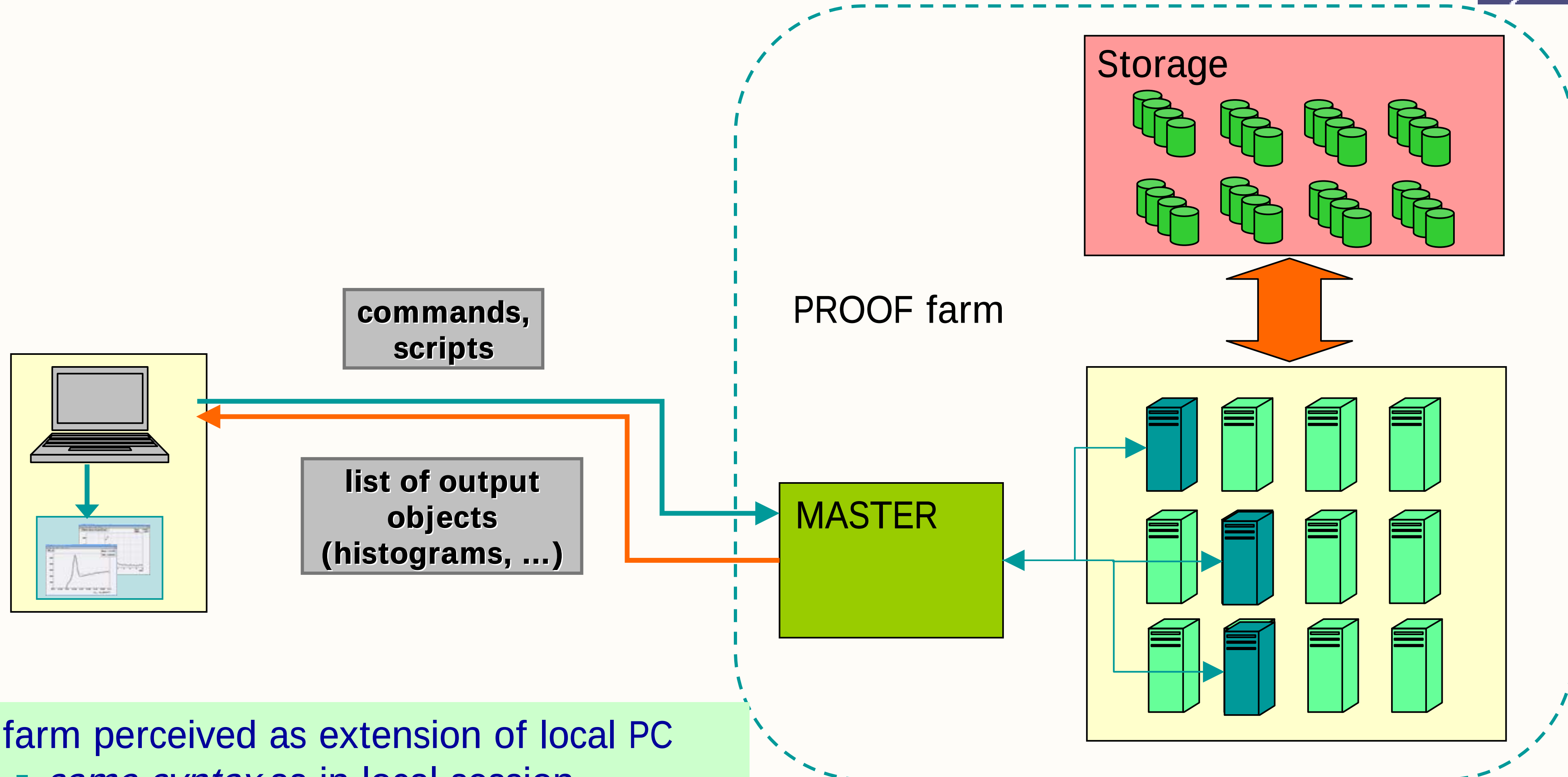








PROOF



- farm perceived as extension of local PC
  - *same syntax* as in local session
- more dynamic use of resources
- real time feedback
- automated splitting and merging

ROOT: An Open Source Project

- Started in 1995
- 11 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

Let's fire up ROOT!

Setting Up ROOT

Before starting ROOT:

setup environment variables $ROOTSYS, $PATH, $LD_LIBRARY_PATH

Go to where ROOT is:

```
$ cd /path-to/root
```

(ba)sh:

```
$ . bin/thisroot.sh
```

(t)csh:

```
$ source bin/thisroot.csh
```

Starting Up ROOT

ROOT is prompt-based.

```
$ root
root [0] _
```

Prompt speaks C++

```
root [0] gROOT->GetVersion()↵
(const char* 0x5ef7e8)"5.16/00"
```

ROOT As Pocket Calculator

Calculations:

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

Running Code

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()
```

All of CINT's commands (help):

```
root [0] .h
```

CINT Interface

Highlights

- Debugger examples
  - show execution: `.T`
  - show TObject: `.class TObject`
  - next statement: `.s`
- Optimizer, to turn off optimization, needed for some loops: `.O0` (dot-oh-zero)
- Redirect output to file: `.> output.txt`
- Quit: `.q`

ROOT Prompt

- ? Why C++ and not a scripting language?!
- ! You'll write your code in C++, too. Support for python, ruby,… exists.

- ? Why a prompt instead of a GUI?
- ! ROOT is a programming framework, not an office suite. Use GUIs where needed.

Compiler, libraries, what's known at the prompt:

Still to come!

Running Code

Macro: file that is interpreted by CINT (`.x`)

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

Execute with `.x mymacro.C(42)`

Unnamed Macros

No functions, just statements

```
{
   float ret = 0.42;
   return sin(ret);
}
```

Execute with `.x mymacro.C`

- No functions thus no arguments

Recommend named macros!

- Compiler prefers it, too…

Named Vs. Unnamed

Named: **function scope**

mymacro.C:

```
void mymacro()
{ int val = 42; }
```

Back at the prompt:

```
root [] val
Error: Symbol val
is not defined
```

unnamed: **global**

unnamed.C:

```
{ int val = 42; }
```

```
root [] val
(int)42
```

Real Survivors: Heap Objects

obj local to function, inaccessible from outside:

```
void mymacro()
{ MyClass obj; }
```

Instead: create on heap, pass to outside:

```
MyClass* mymacro()
{ MyClass* pObj = new MyClass();
  return pObj; }
```

pObj gone – but MyClass still where pObj pointed to! Returned by mymacro()

Running Code – Libraries

"Library": compiled code, shared library

CINT can call its functions!

Build a library: ACLiC! "+" instead of Makefile
(**A**utomatic **C**ompiler of **Li**braries for **C**INT)

```
.x something.C+(42)
```

CINT knows all its functions / types:

```
something(42)
```

Compiled versus Interpreted

? Why compile?

! Faster execution, CINT has limitations…

? Why interpret?

! Faster Edit → Run → Check result → Edit cycles ("rapid prototyping").
Scripting is sometimes just easier.

? Are Makefiles dead?

! Yes! ACLiC is even platform independent!

Summary

We know:

- why and how to start ROOT
- that you run your code with ".x"
- can call functions in libraries
- can (mis-) use ROOT as a pocket calculator!

Lots for you to discover during next three lectures and especially the exercises!

Saving Data

Streaming, Reflection, TFile, Schema Evolution

Saving Objects – the Issues

```
long aLong = 42;
```

WARNING:
platform
dependent!

Storing aLong:

- How many bytes? Valid:
  0x0000002a
  0x000000000000002a
  0x0000000000000000000000000000002a
- which byte contains 42, which 0? Valid:
  char[] {42, 0, 0, 0}: little endian, e.g. Intel
  char[] {0, 0, 0, 42}: big endian, e.g. PowerPC

Platform Data Types

Fundamental data types:

- size is platform dependent

Store "int" on platform A

- 0x000000000000002a

Read back on platform B – incompatible!

- Data loss, size differences, etc:
- 0x00000000~~0000002a~~

ROOT Basic Data Types

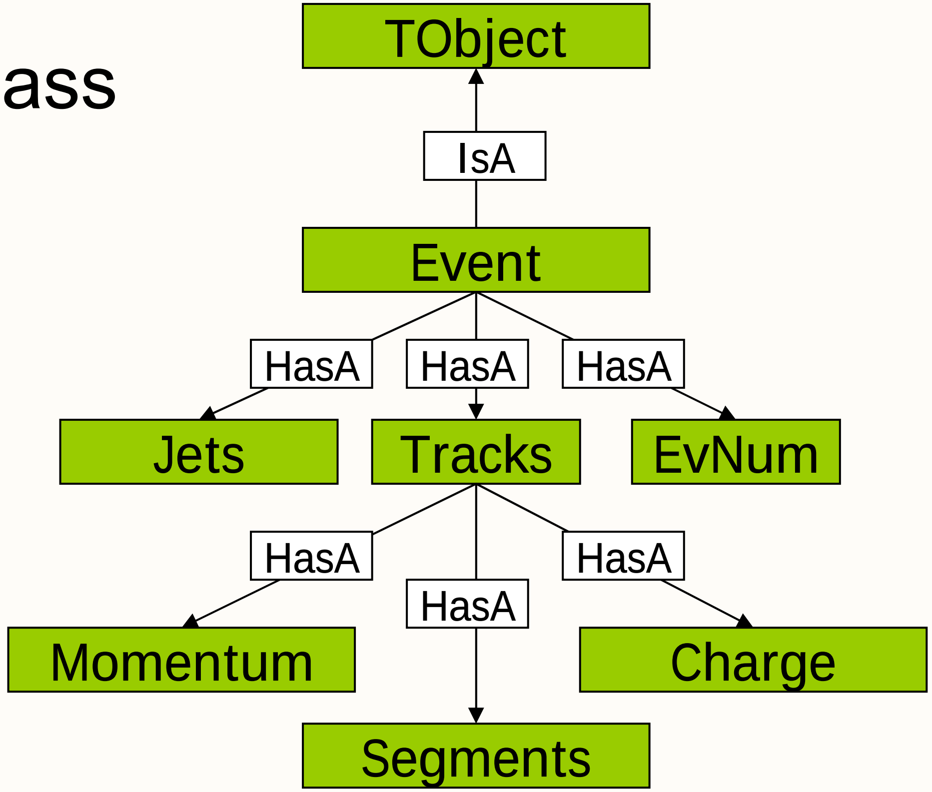
Solution: ROOT typedefs

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

Object Oriented Concepts

- Class: the description of a “thing” in the system
- Object: instance of a class
- Methods: functions for a class

- Members: a “has a” relationship to the class.
- Inheritance: an “is a” relationship to the class.



Saving Objects – the Issues

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

WARNING:
platform
dependent!

Storing **`anObject`**:

- need to know its members + base classes
- need to know "where they are"

Reflection

Need type description (aka *reflection*)

1. types, sizes, members

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

TMyClass is a class.

Members:

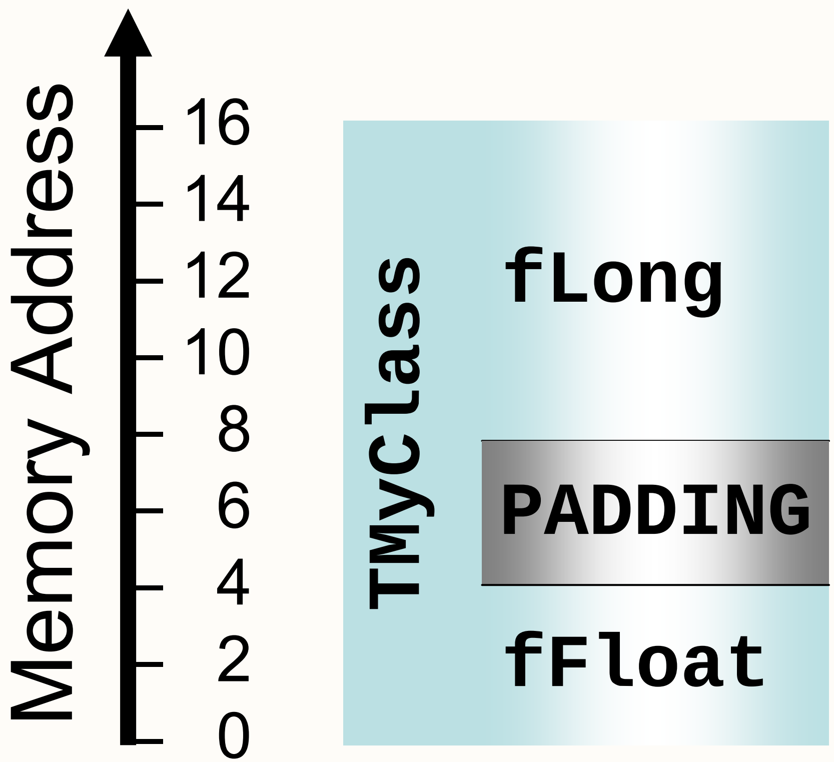
- "fFloat", type float, size 4 bytes
- "fLong", type Long64_t, size 8 bytes

Reflection

Need type description

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

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

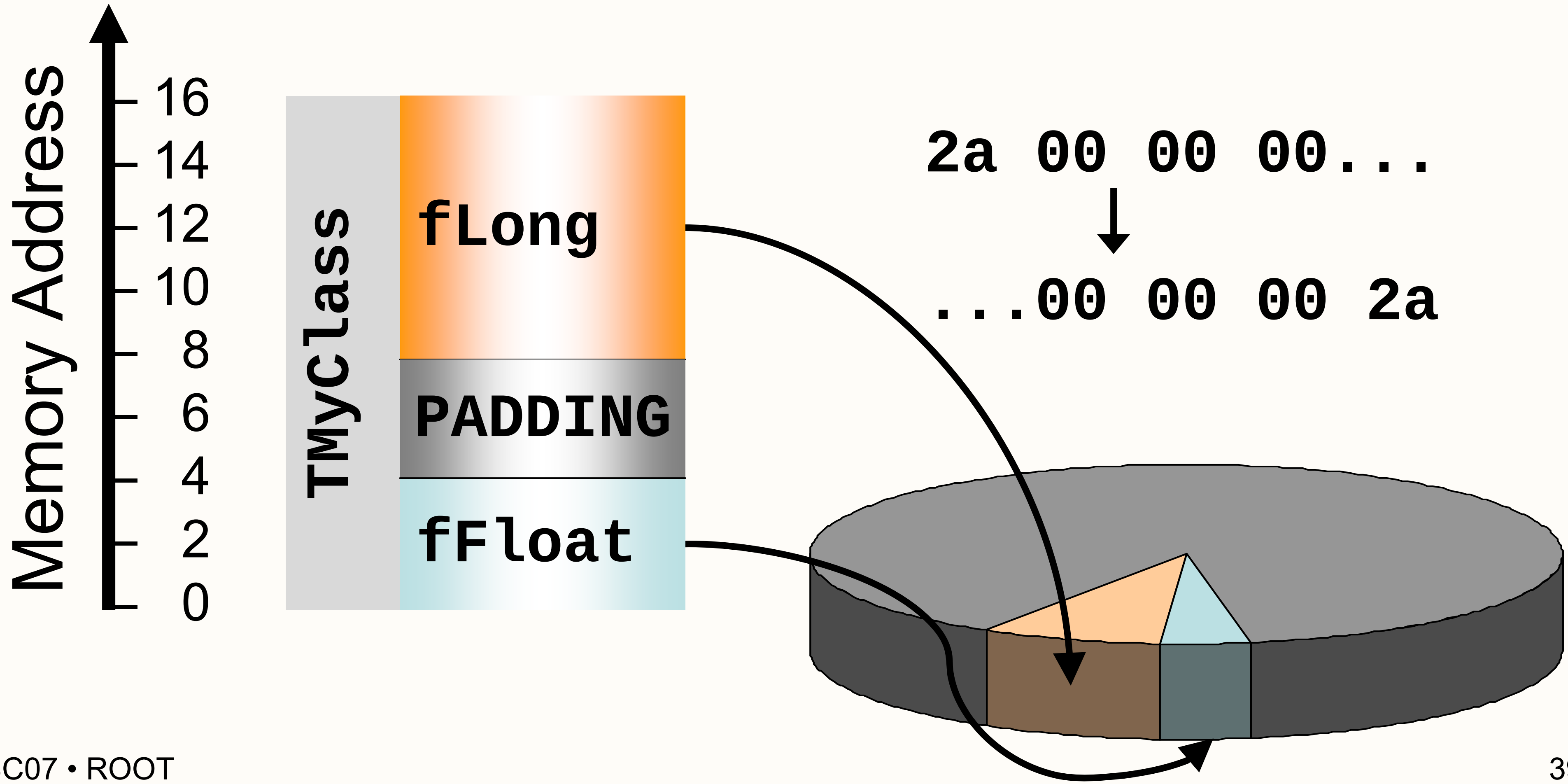


"fFloat" is at offset 0

"fLong" is at offset 8

I/O Using Reflection

members → memory → re-order →



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++.

BE CAREFUL

THIS LANGUAGE HAS NO BRAIN USE YOUR OWN

Reflection For C++

Program parses header files, e.g. Funky.h:

```
rootcint –f Dict.cxx –c Funky.h LinkDef.h
```

Collects reflection data for types requested in Linkdef.h

Stores it in Dict.cxx (dictionary file)

Compile Dict.cxx, link, load:
C++ with reflection!

Reflection By Selection

LinkDef.h syntax:

```
#pragma link C++ class MyClass+;
#pragma link C++ typedef MyType_t;
#pragma link C++ function MyFunc(int);
#pragma link C++ enum MyEnum;
```

Why So Complicated?

Simply use ACLiC:

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

Will create library with dictionary
of all types, namespaces, functions
declared in MyCode.cxx, MyCode.h/.hpp,…
automatically!

Back To Saving Objects: TFile

ROOT stores objects in TFiles:

```
TFile* f = new TFile("afile.root", "NEW");
```

TFile behaves like file system:

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

TFile has a current directory:

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

Interlude: HELP!

What is a TFile?

What functions does it have?

Documentation!

User's Guide (it has your answers!):

http://root.cern.ch

Reference Guide (class documentation):

http://root.cern.ch/root/html

Saving Objects, Really

Given a TFile:

```
TFile* f = new TFile("afile.root", "RECREATE");
```

Write an object deriving from TObject:

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

"optionalName" or **`TObject::GetName()`**

Write any object (with dictionary):

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

"Where Is My Histogram?"

TFile owns histograms, graphs, trees (due to historical reasons):

```
TFile* f = new TFile("myfile.root");
TH1F* h = new TH1F("h","h",10,0.,1.);
h->Write();
Canvas* c = new TCanvas();
c->Write();
delete f;
```

h automatically deleted: owned by file.

c still there.

TFile as Directory: TKeys

One TKey per object:

- named directory entry
- knows what it points to
- like inodes in file system

Each TFile directory is collection of TKeys

TFile "myfile.root"

- TKey: "hist1", TH1F
- TKey: "list", TList
- TKey: "hist2", TH2F
- TDirectory: "subdir"
  - TKey: "hist1", TH2D

TKeys Usage

Efficient for hundreds of objects

Inefficient for millions:

- lookup ≥ log(n)
- sizeof(TKey) on 64 bit machine: 160 bytes

Better storage / access methods available,
see next lecture

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

MyObject;2

MyObject;3

…

MyObject

Write() does not replace but append!

but see documentation TObject::Write()

The "I" Of I/O

Reading is simple:

```
TFile* f = new TFile("myfile.root");
TH1F* h = 0;
f->GetObject("h", h);
h->Draw();
delete f;
```

Remember:

- TFile owns histograms!
- file gone, histogram gone!

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.

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!

Objects written with old version cannot be read!

Solution: Schema Evolution

Support changes:

1. removed members: just skip data

file.root		RAM
`Long64_t fLong;`	ignore	
`float fFloat;`	→	`float fFloat;`

2. added members: just default-initialize them (whatever the default constructor does)

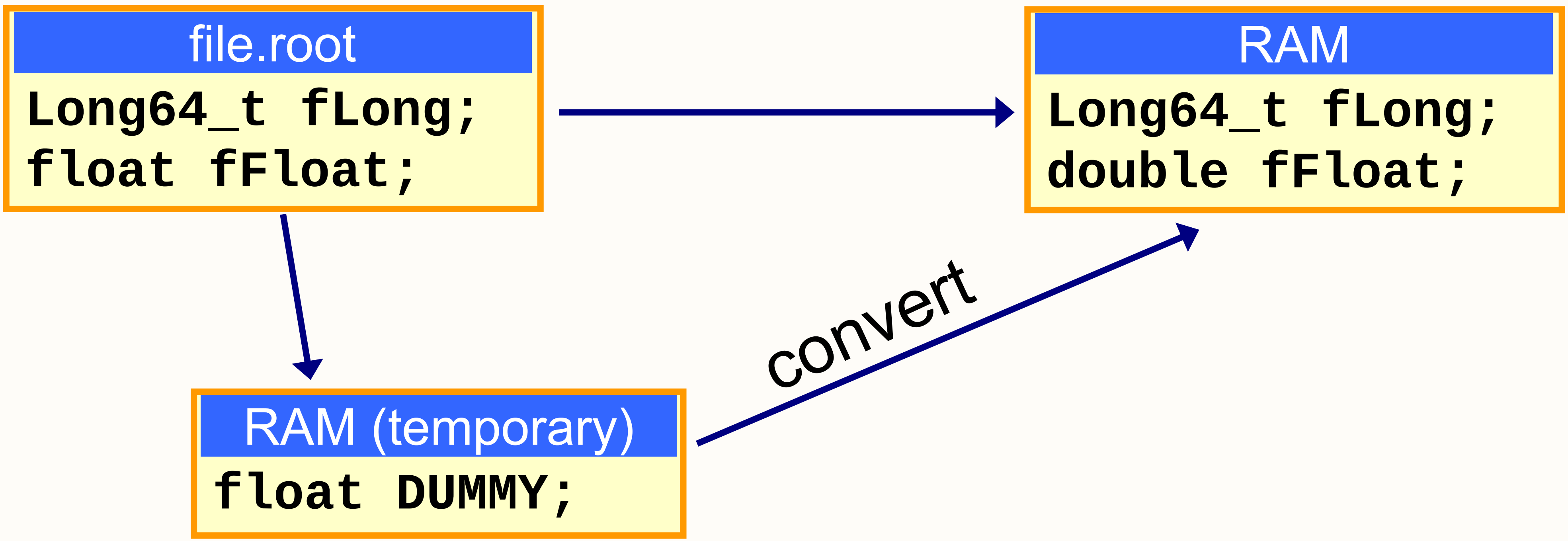
`TMyClass(): fLong(0)`

file.root		RAM
	`TMyClass(): fLong(0)` →	`Long64_t fLong;`
`float fFloat;`	→	`float fFloat;`

Solution: Schema Evolution

Support changes:

3. members change types



Supported Type Changes

Schema Evolution supports:

- float ↔ double ↔ int ↔ long, etc.
- float* ↔ double* ↔ int* ↔ long*, etc.
- TYPE* ↔ std::vector<TYPE>
- CLASS* ↔ TClonesArray

Adding / removing base class equivalent to adding / removing data member

Storing Reflection

Big Question: file == memory class layout?

Need to store reflection to file, see TFile::ShowStreamerInfo():

```
StreamerInfo for class: TH1F, version=1
  TH1     BASE    offset=  0 type= 0
  TArrayF BASE    offset=  0 type= 0

StreamerInfo for class: TH1, version=5
  TNamed  BASE    offset=  0 type=67
...
  Int_t   fNcells offset=  0 type= 3
  TAxis   fXaxis  offset=  0 type=61
```

Class Version

- *One* TStreamerInfo for *all* objects of the same class layout in a file
- Can have multiple versions in same file
- Use version number to identify layout:

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

Random Facts On ROOT I/O

- We know exactly what's in a TFile – need no library to look at data! (Examples tomorrow.)
- ROOT files are zipped
- Combine contents of TFiles with **`$ROOTSYS/bin/hadd`**
- Can even open **`TFile("http://myserver.com/afile.root")`** including read-what-you-need!
- Nice viewer for TFile: **`new TBrowser`**

Summary

Big picture:

- you know ROOT files – for petabytes of data
- you learned what schema evolution is
- you learned that reflection is key for I/O

Small picture:

- you can write your own data to files
- you can read it back
- you can change the definition of your classes

ROOT Collection Classes

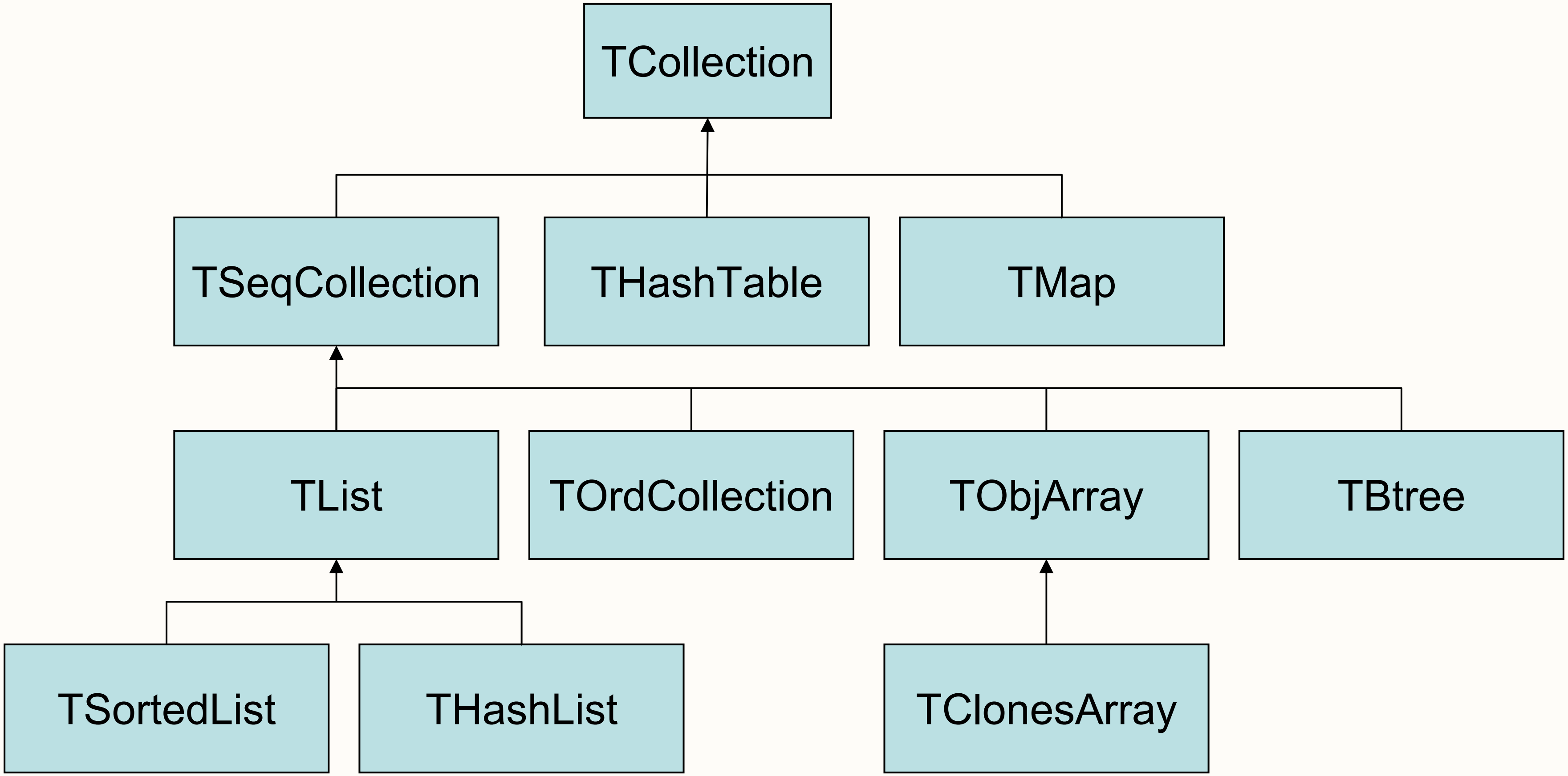
Or when one million TKeys are not enough…

Collection Classes

The ROOT collections are polymorphic containers that hold pointers to TObjects, so:

- They can only hold objects that inherit from **TObject**
- They return pointers to **TObjects**, that have to be cast back to the correct subclass

Types Of Collections



The inheritance hierarchy of the primary collection classes

Collections (cont)

Here is a subset of collections supported by ROOT:

- TObjArray
- TClonesArray
- THashTable
- THashList
- TMap
- Templated containers, e.g. STL (std::list etc)

TObjArray

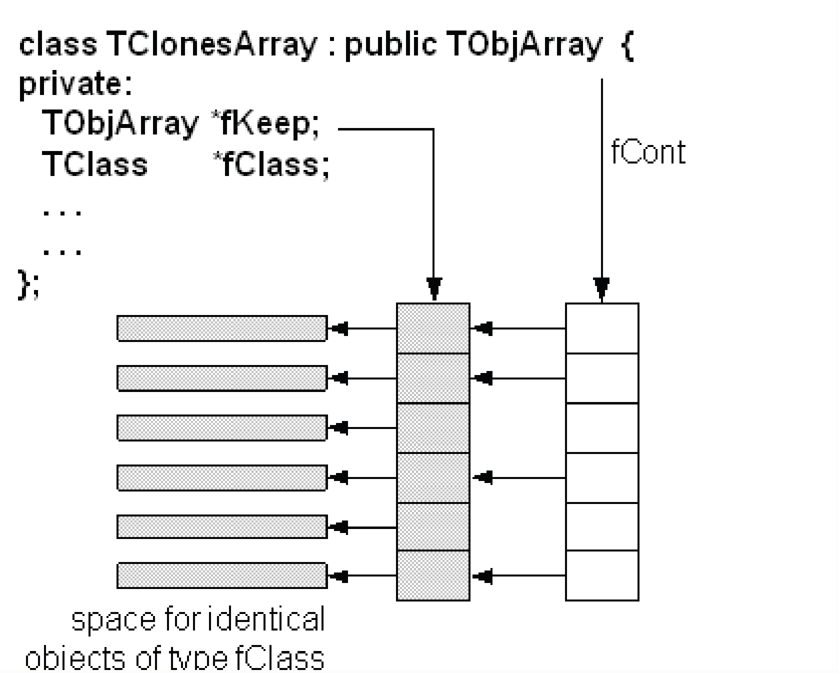
- A **TObjArray** is a collection which supports traditional array semantics via the overloading of operator[].
- Objects can be directly accessed via an index.
- The array expands automatically when objects are added.

TClonesArray

Array of identical classes ("clones").

Designed for repetitive data analysis tasks: same type of objects created and deleted many times.

No comparable class in STL!



The internal data structure of a TClonesArray

Traditional Arrays

Very large number of new and delete calls in large loops like this (N(100000) x N(10000) times new/delete):

N(100000)

```
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();
}
```

N(10000)

You better use a **TClonesArray** which reduces the number of new/delete calls to only N(10000):

```
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(100000)

N(10000)

Pair of new/delete calls cost about 4 μs

NN(10^9) new/deletes will save about 1 hour.

THashTable - THashList

THashTable puts objects in one of several of its short lists. Which list to pick is defined by TObject::Hash(). Access much faster than map or list, but no order defined. Nothing similar in STL.

THashList consists of a THashTable for fast access plus a TList to define an order of the objects.

TMap

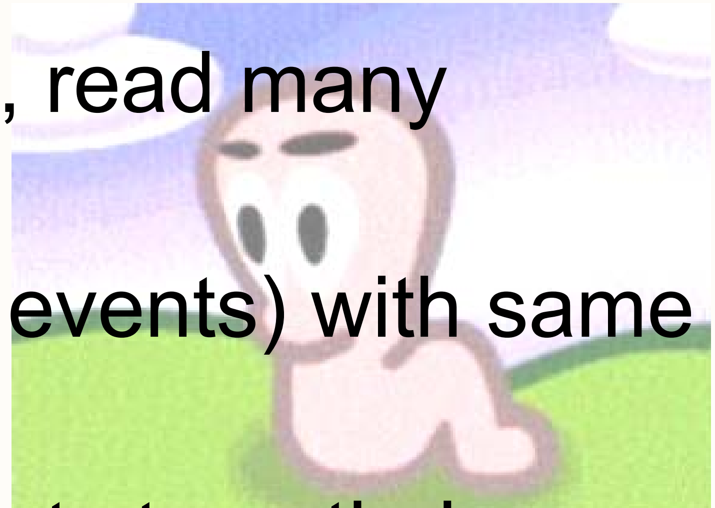
TMap implements an associative array of (key,value) pairs using a **THashTable** for efficient retrieval (therefore **TMap** does not conserve the order of the entries).

ROOT Trees

Why Trees ?

- As seen previously, any object deriving from TObject can be written to a file with an associated key with object.Write()
- However each key has an overhead in the directory structure in memory (up to 160 bytes). Thus, Object.Write() is very convenient for simple objects like histograms, but not for saving one object per event.
- Using TTrees, the overhead in memory is in general less than 4 bytes per entry!

Why Trees ?

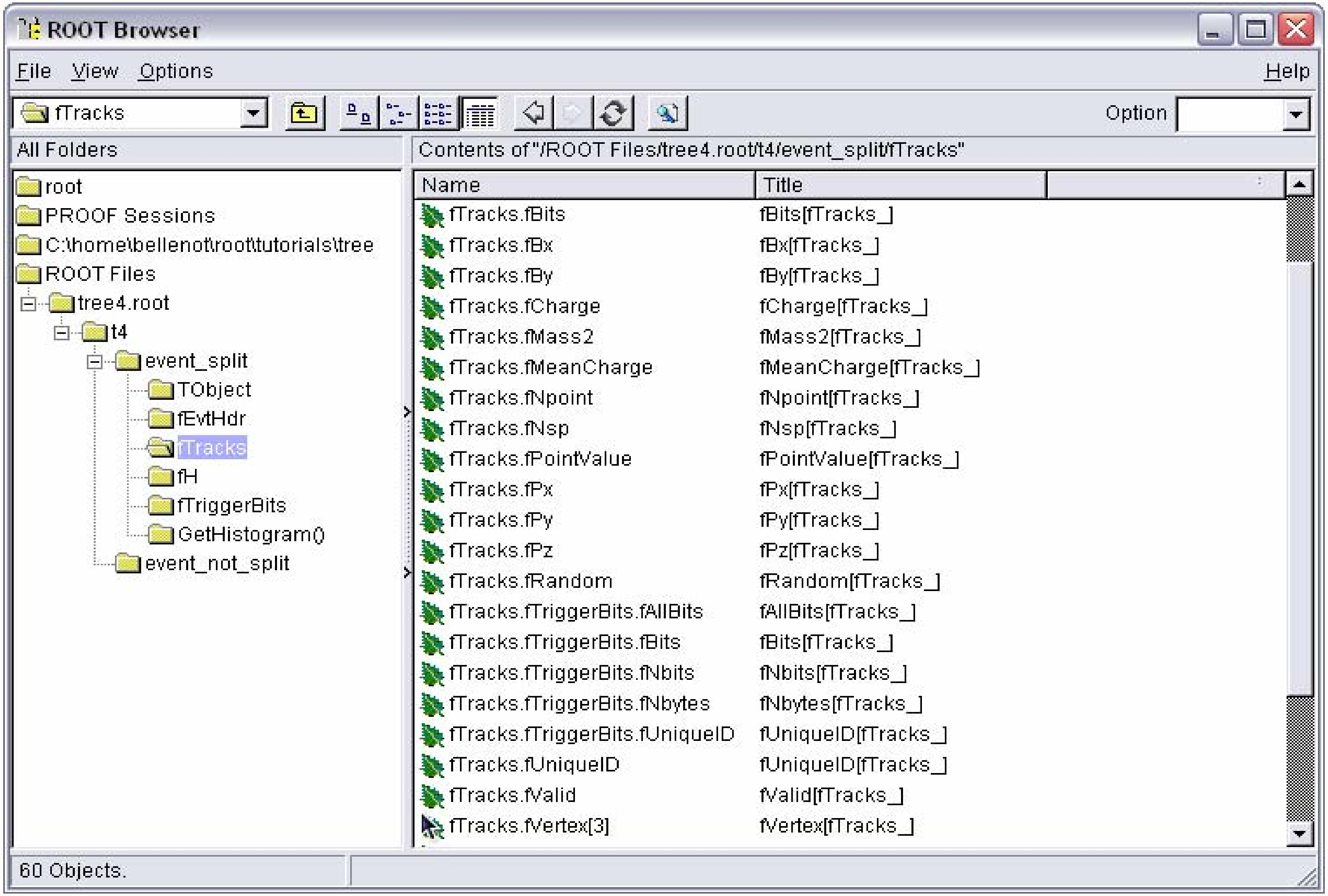


- Extremely efficient write once, read many ("WORM")
- Designed to store $>10^9$ (HEP events) with same data structure
- Load just a subset of the objects to optimize memory usage
- Trees allow fast direct and random access to any entry (sequential access is the best)

Building ROOT Trees

- Overview of
  - Trees
  - Branches
- 5 steps to build a TTree

Tree structure

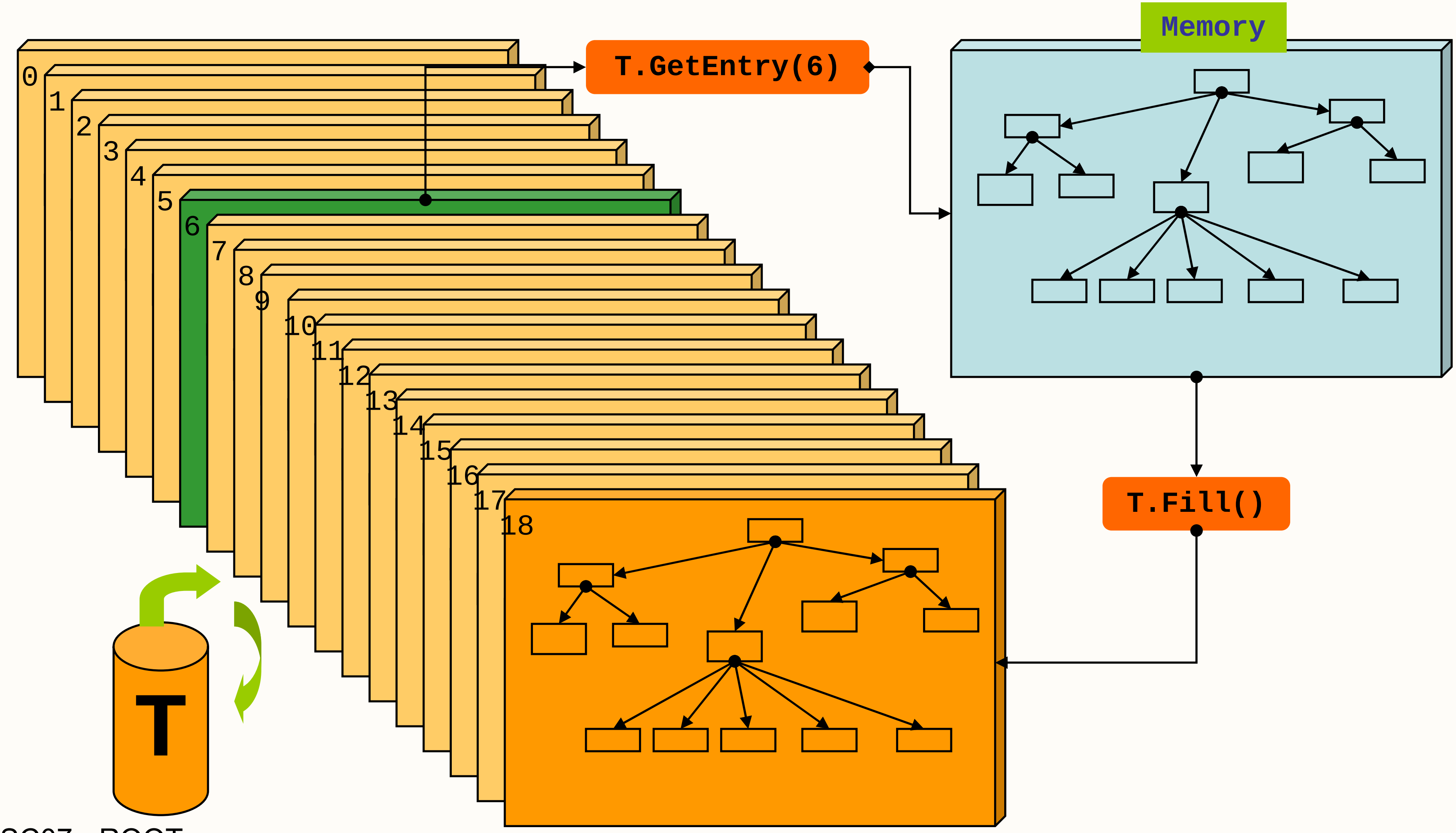


Tree structure

- Branches: directories
- Leaves: data containers
- Can read a subset of all branches – speeds up considerably the data analysis processes
- Tree layout can be optimized for data analysis
- The class for a branch is called **TBranch**
- Variables on **TBranch** are called leaf (yes - **TLeaf**)
- Branches of the same **TTree** can be written to separate files

Memory ↔ Tree

Each Node is a branch in the Tree



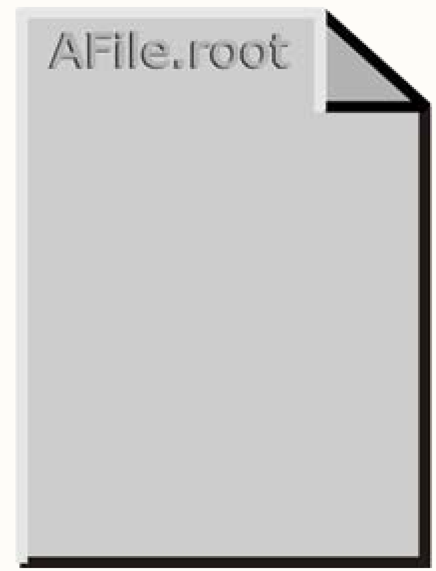
Five Steps to Build a Tree

Steps:

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

Step 1: Create a TFile Object

Trees can be huge → need file for swapping filled entries

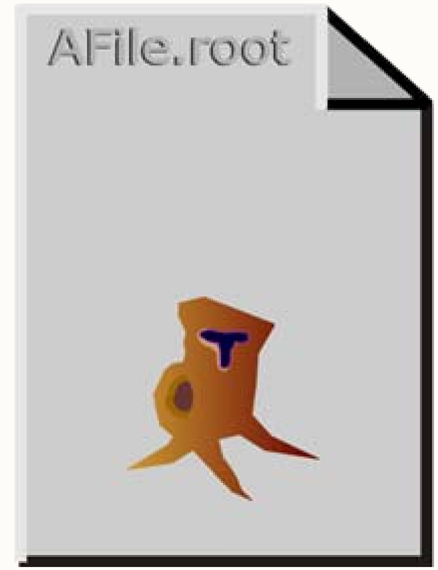



```
TFile *hfile = new TFile("AFile.root");
```

Step 2: Create a TTree Object

The TTree constructor:

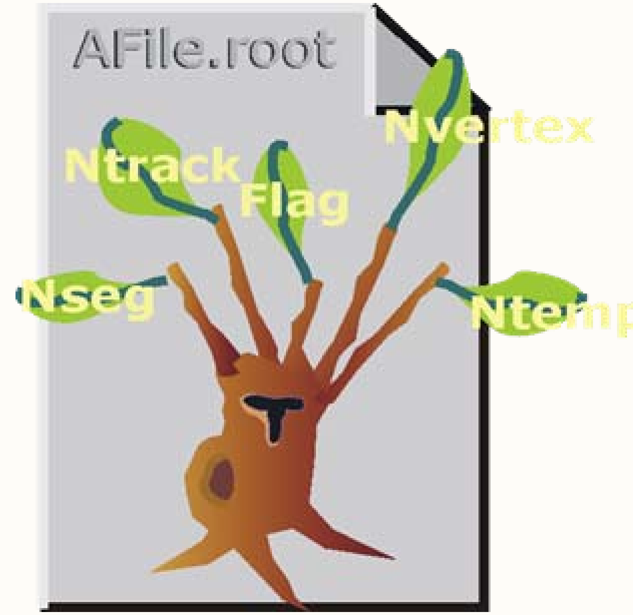
- Tree name (e.g. "myTree")
- Tree title




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

Step 3: Adding a Branch

- Branch name
- <u>Address of the pointer</u> to the object

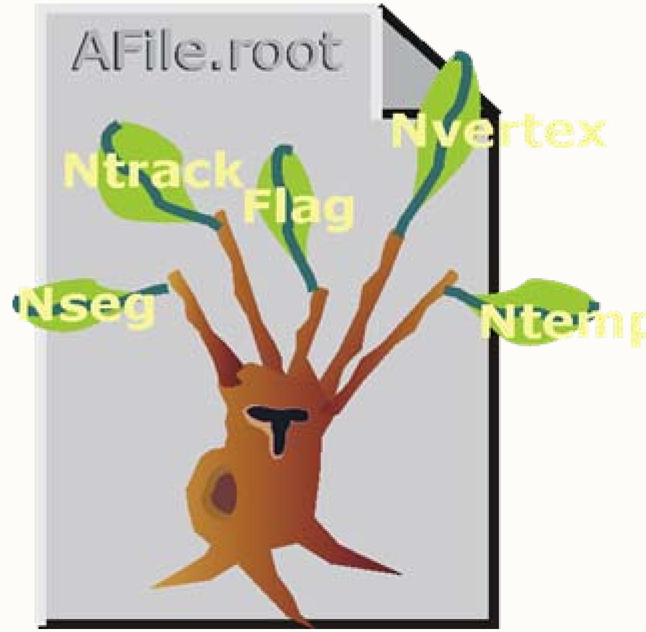



```
Event *event = new  Event();
myTree->Branch("EventBranch", &event);
```

Splitting a Branch

Setting the split level (default = 99)

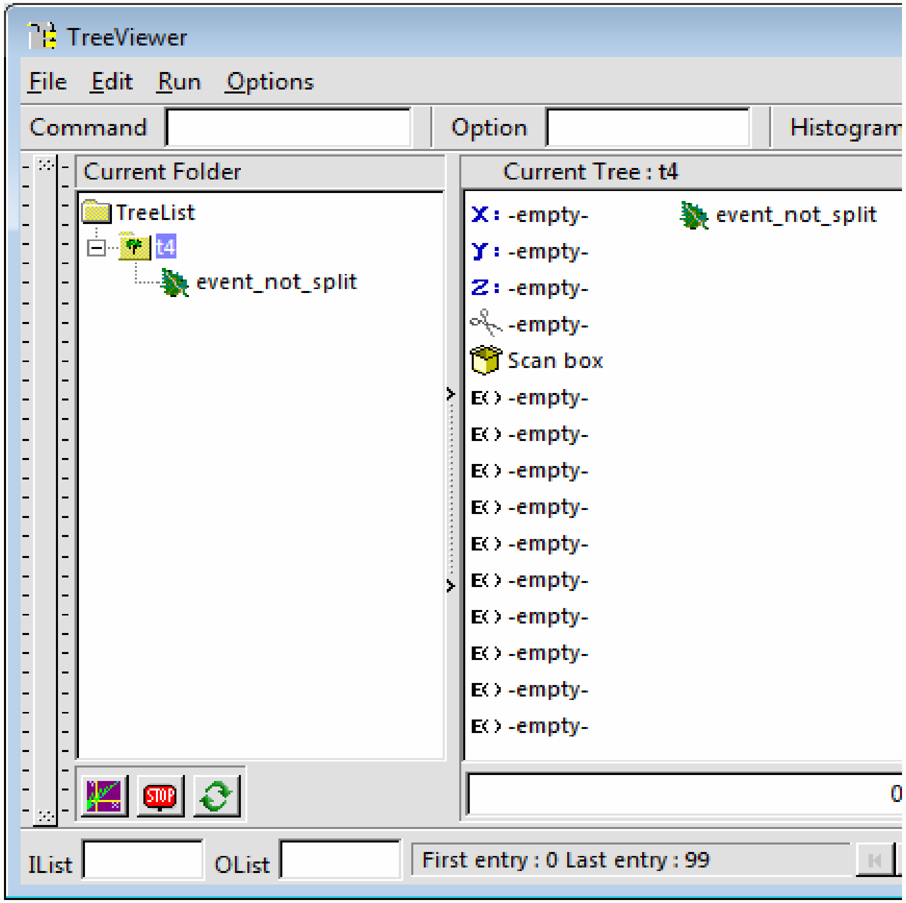
Split level = 0



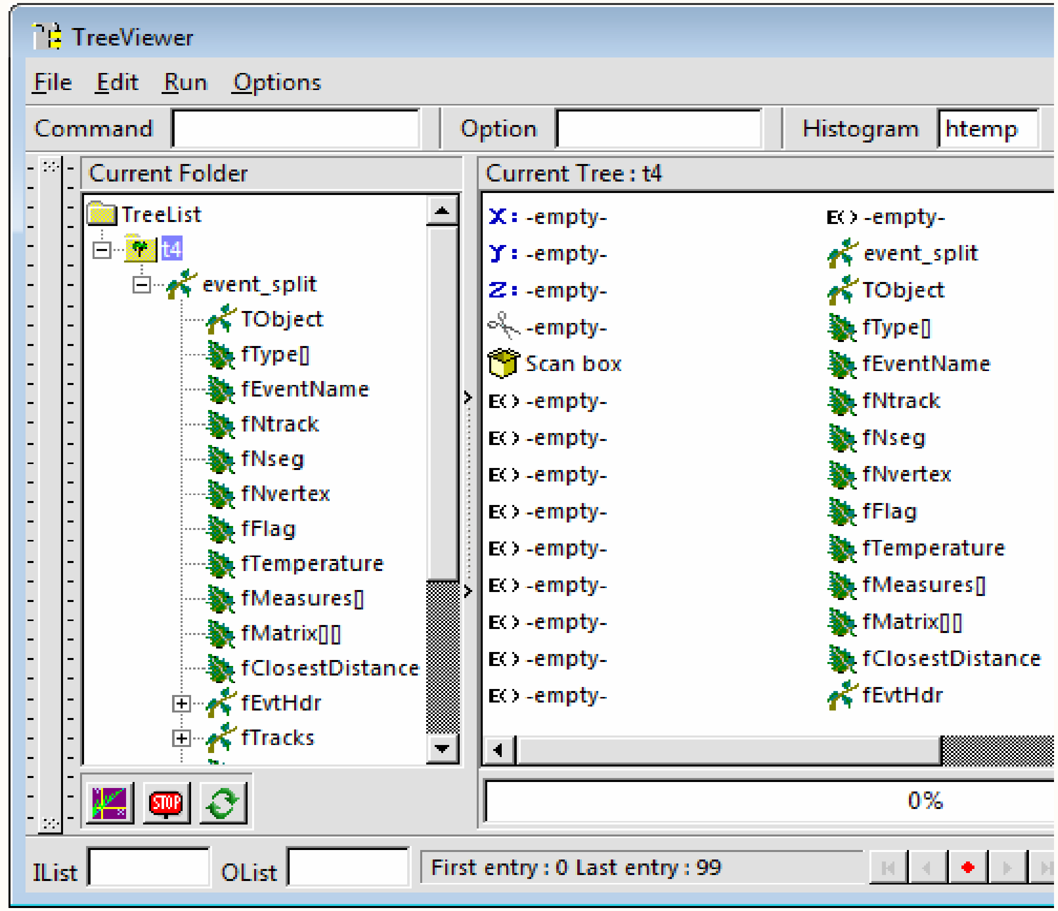
Split level = 99

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

Splitting (real example)



Split level = 0



Split level = 99

Splitting

- 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, instead of full object
- Supports STL containers, too!

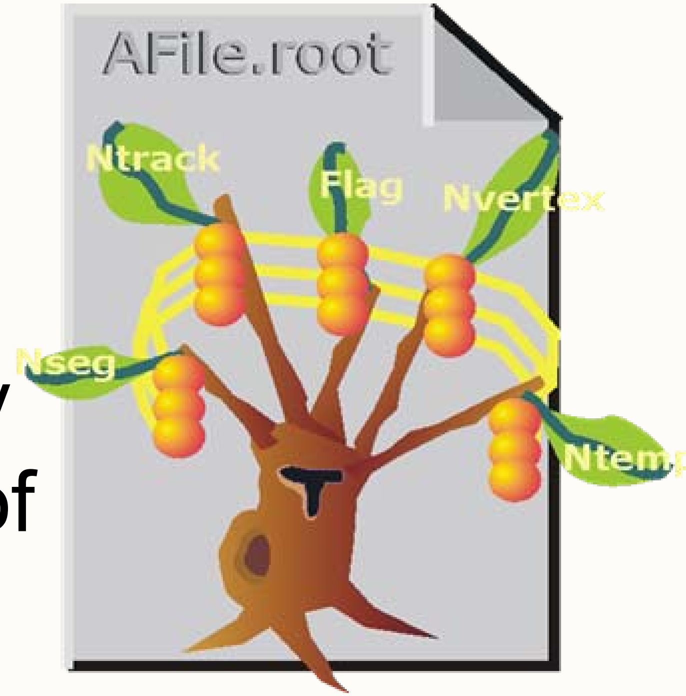
Performance Considerations

A split branch is:

- Faster to read - the type does not have to be read each time
- Slower to write due to the large number of buffers

Step 4: Fill the Tree

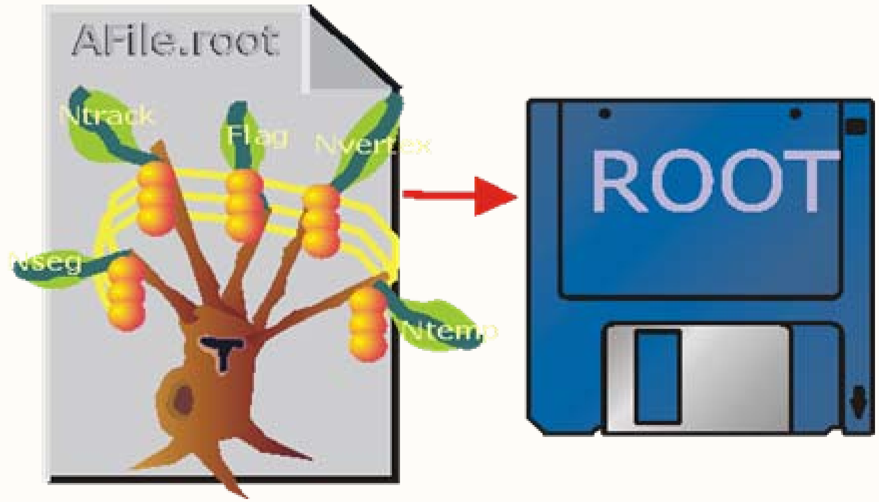
- Create a for loop
- Assign values to the object contained in each branch
- TTree::Fill() creates a new entry in the tree: snapshot of values of branches' objects




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

Step 5: Write Tree To File

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



Example macro

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

Reading a TTree

- Looking at a tree
- How to read a tree
- Trees, friends and chains

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  :EventBranch                                                        *
*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     *
*............................................................................*
…
…
```

Looking at the Tree

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 *
```

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 exactly 0.

Looking at the Tree

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

```
 root [] myTree->Show(0);
======> EVENT:0
 EventBranch      = 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 TTree

Example:

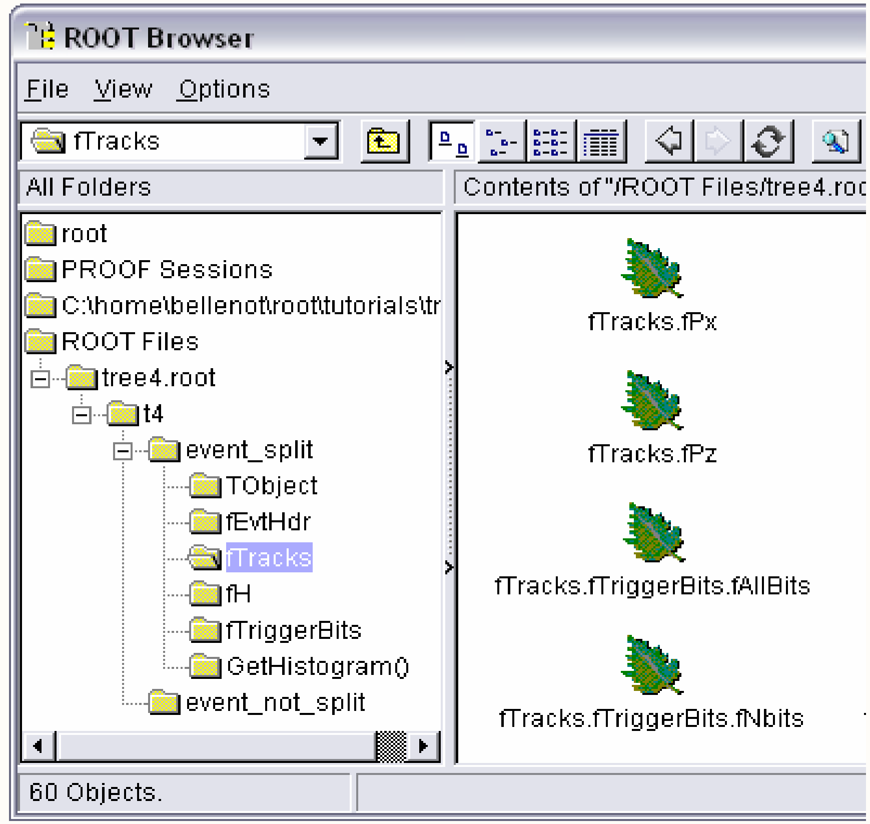
1. Open the Tfile

```
TFile f("tree4.root")
```

2. Get the TTree

```
TTree *t4=0;
f.GetObject("t4",t4)
```

OR



How to Read a TTree

3. Create a variable pointing to the data

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

4. Associate a branch with the variable:

```
root [] t4->SetBranchAddress("event_split", &event);
```

5. Read one entry in the TTree

```
root [] t4->GetEntry(0)
root [] event->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
[...]
```

Selecting Branches For Reading

By default, TTree::GetEntry() reads all branches

Can select subset by disabling all:

```
MyTree->SetBranchStatus("*", 0);
```

and the re-enabling the branches to be read:

```
MyTree->SetBranchStatus("branch1", 1);
MyTree->SetBranchStatus("branch2.subbranch*", 1);
```

Example macro

```
{
  Event *ev = 0;
  TFile f("mytree.root");
  TTree *myTree = (TTree*)f->Get("myTree");
  myTree->SetBranchAddress("SplitEvent", &ev);
  for (int e=0;e<100000;++e) {
     myTree->GetEntry(e);
     ev->Analyse();
  }
}
```

TChain: the Forest

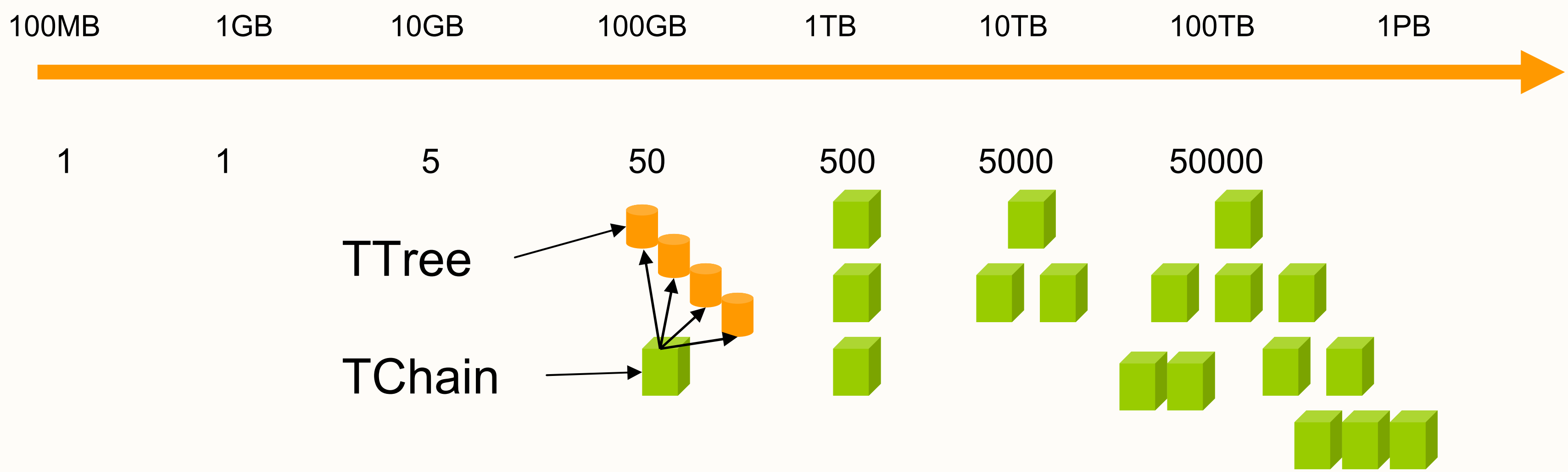
- Collection of TTrees: list of ROOT files containing the same tree
- Same semantics as TTree

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");
```

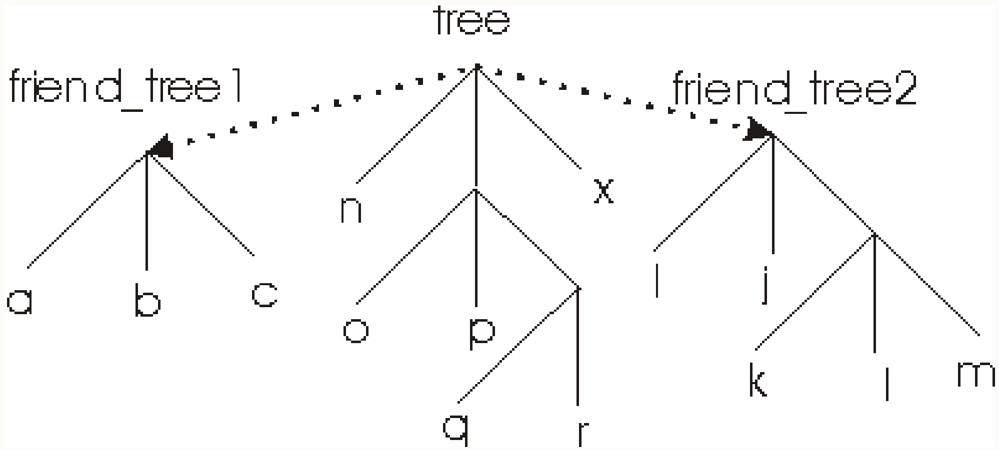
Now we can use the TChain like a TTree!

Data Volume & Organisation



- A TFile typically contains 1 TTree
- A TChain is a collection of TTrees or/and TChains
- A TChain is typically the result of a query to a file catalog

Friends of Trees

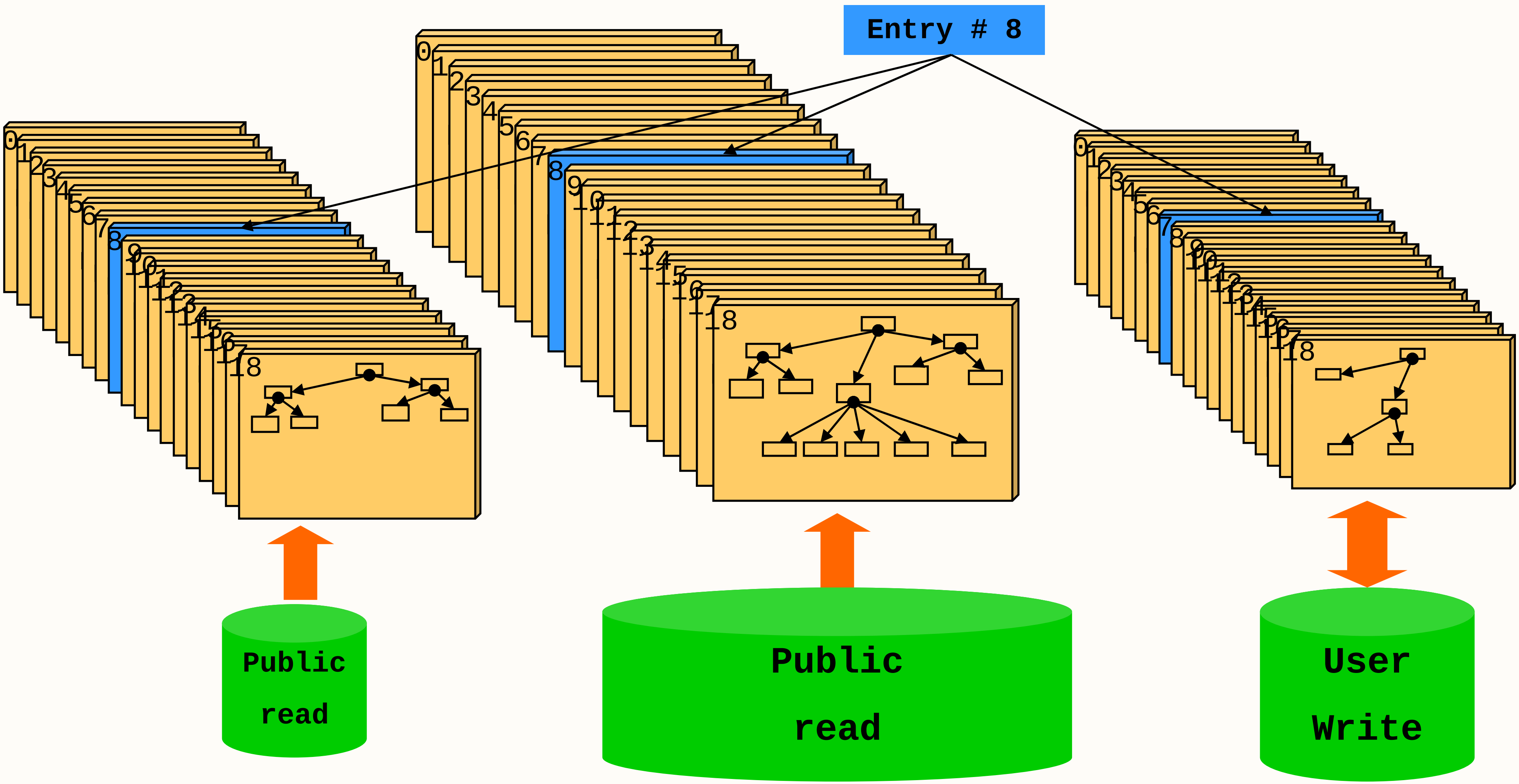


- Adding new branches to existing tree without touching it, i.e.:

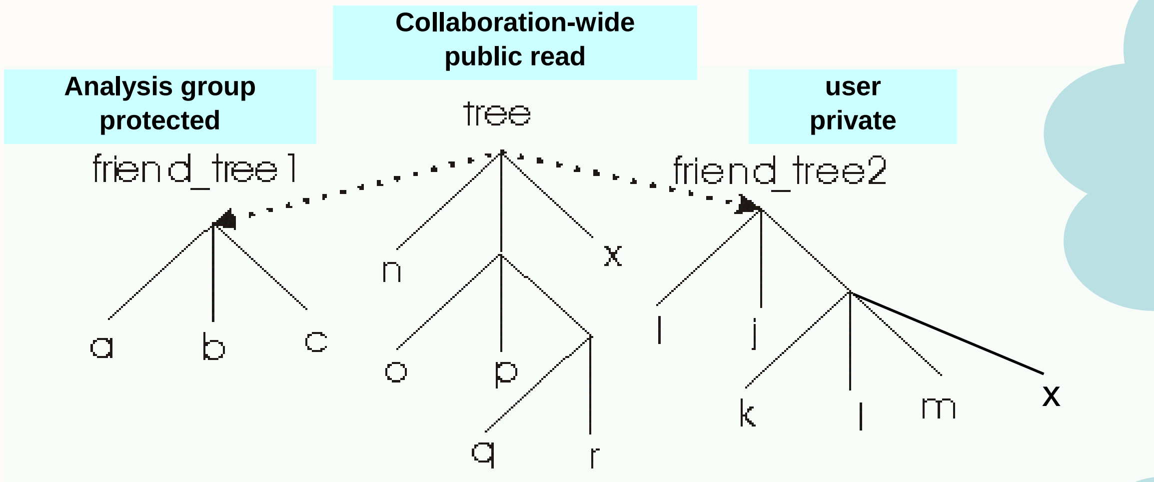
```
myTree->AddFriend("ft1", "friend.root")
```

- Unrestricted access to the friend's data via virtual branch of original tree

Tree Friends



Tree Friends



Processing time independent of the number of friends unlike table joins in RDBMS

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

Summary: Reading Trees

- TTree 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 (WORM) ideal for experiments' data
- Still: extensible, users can add their own tree as friend

Analyzing Trees

Selectors, Proxies, PROOF

Recap

TTree efficient storage and access for huge amounts of structured data

Allows selective access of data

TTree knows its layout

Almost all HEP analyses based on TTree

TTree Data Access

Data access via TTree / TBranch is complex

Lots of code identical for all TTrees:
  getting tree from file, setting up branches, entry loop

Lots of code completely defined by TTree:
  branch names, variable types, branch ↔ variable mapping

Need to enable / disable branches "by hand"

Want common interface to allow generic "TTree based analysis infrastructure"

TTree Proxy

The solution:

- write analysis code using branch names as variables
- auto-declare variables from branch structure
- read branches on-demand

Branch vs. Proxy

Take a simple branch:

```
class ABranch {
  int a;
};
ABranch* b;
tree->Branch("abranch", &b);
```

Access via proxy in pia.C:

```
double pia() {
  return sqrt(abranch.a * TMath::Pi());
}
```

Proxy Details

```
double pia() {
  return sqrt(abranch.a * TMath::Pi());
}
```

Analysis script somename.C must contain somename()

Put #includes into somename.h

Return type must convert to double

Proxy Advantages

```
double pia() {
  return sqrt(abranch.a * TMath::Pi());
}
```

Very efficient: only reads leaf "a"

Can use arbitrary C++

Leaf behaves like a variable

Uses meta-information stored with TTree:

- branch names
- types contained in branches / leaves
- and their members (unrolling)

Simple Proxy Analysis

TTree::Draw() runs simple analysis:

```
TFile* f = new TFile("tree.root");
TTree* tree = 0;
f->GetObject("MyTree", tree);
tree->Draw("pia.C+");
```

Compiles pia.C

Calls it for each event

Fills histogram named "htemp" with return value

Behind The Proxy Scene

TTree::Draw("pia.C+") creates helper class

```
generatedSel: public TSelector {
#include "pia.C"
  ...
};
```

pia.C gets **`#included`** inside generatedSel!

Its data members are named like leaves, wrap access to Tree data

Can also generate Proxy via

```
tree->MakeProxy("MyProxy", "pia.C")
```

TSelector

```
generatedSel: public TSelector
```

TSelector is base for event-based analysis:

1. setup

   ```
   TSelector::Begin()
   ```

2. analyze an entry

   ```
   TSelector::Process(Long64_t entry)
   ```

3. draw / save histograms

   ```
   TSelector::Terminate()
   ```

Extending The Proxy

Given Proxy generated for script.C (like pia.C):

looks for and calls

```
void    script_Begin(TTree* tree)
Bool_t  script_Process(Long64_t entry)
void    script_Terminate()
```

Correspond to TSelector's functions

Can be used to set up complete analysis:

- fill several histograms,
- control event loop

Proxy And TClonesArray

TClonesArray as branch:

```
class TGap { float ore; };
TClonesArray* b = new TClonesArray("TGap");
tree->Branch("gap", &b);
```

Cannot loop over entries and return each:

```
double singapore() {
  return sin(gap.ore[???]);
}
```

Proxy And TClonesArray

TClonesArray as branch:

```
class TGap { float ore; };
TClonesArray* b = new TClonesArray("TGap");
tree->Branch("gap", &b);
```

Implement it as part of pia_Process() instead:

```
Bool_t pia_Process(Long64_t entry) {
  Long_t nGaps = gap.GetEntries()
  for (int i=0; i < nGaps; ++i)
    hSingapore->Fill(sin(gap.ore[i]));
  return kTRUE; }
```

Extending The Proxy, Example

Need to declare hSingapore somewhere!

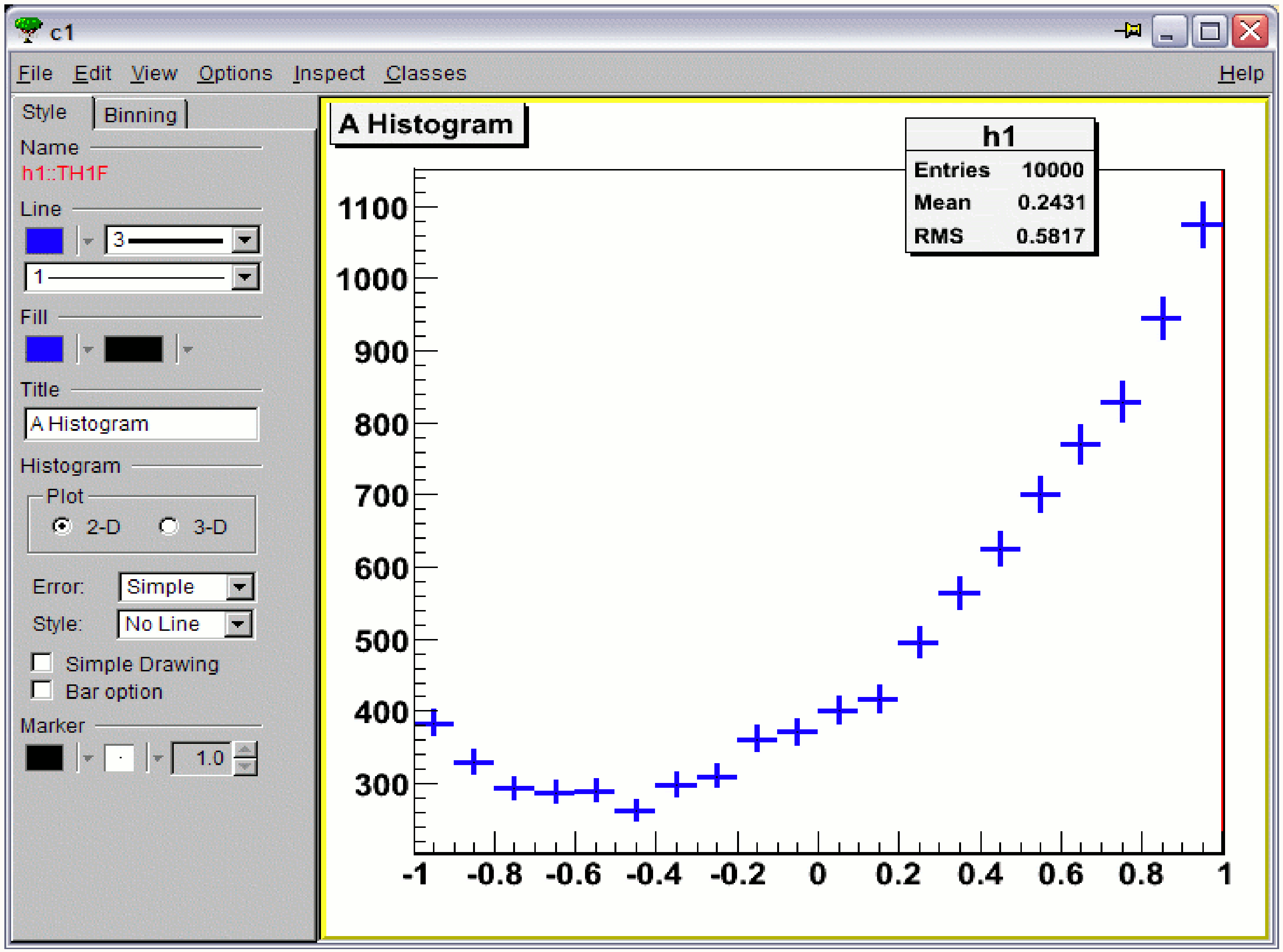
But pia.C is **#included** inside generatedSel, so:

```
TH1* hSingapore;
void pia_Begin() {
  hSingapore = new TH1F(...);
}
Bool_t pia_Process(Long64_t entry) {
  hSingapore->Fill(...);
}
void pia_Terminate() {
  hSingapore->Draw();
}
```

Histograms

Analysis result: often a histogram

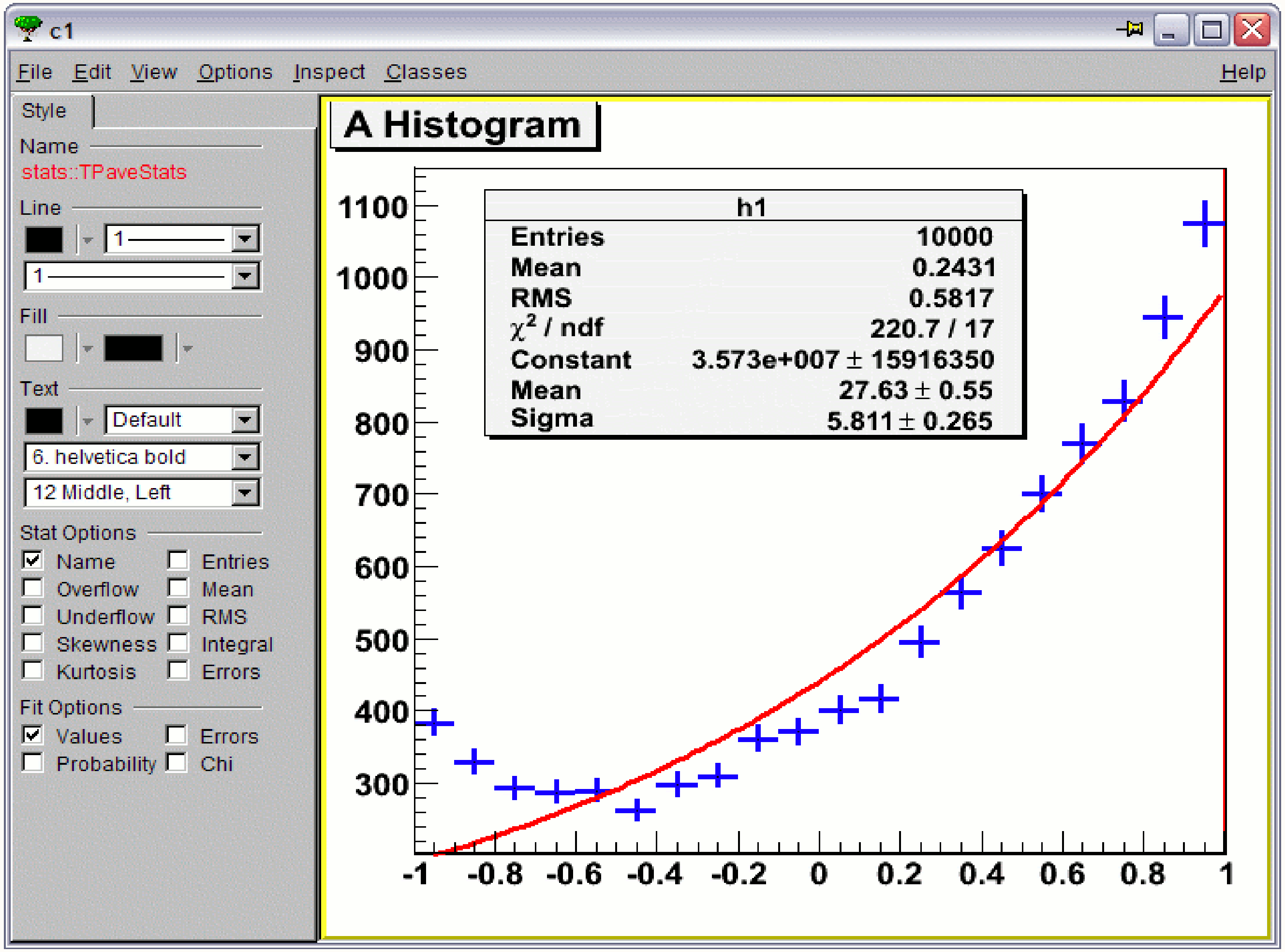
Value (x)
vs. count (y)



Menu:
View / Editor

Fitting

Analysis result:
often a ***fit*** based on a histogram



Fit

Fit = optimization in parameters,

e.g. Gaussian $$f(x) = [0] \cdot e^{-\frac{(x-[1])^2}{2 \cdot [2]^2}}$$

Objective: choose parameters [0], [1], [2] to get function as close as possible to histogram

For Gaussian: [0] = "Constant"
[1] = "Mean"
[2] = "Sigma" / "Width"

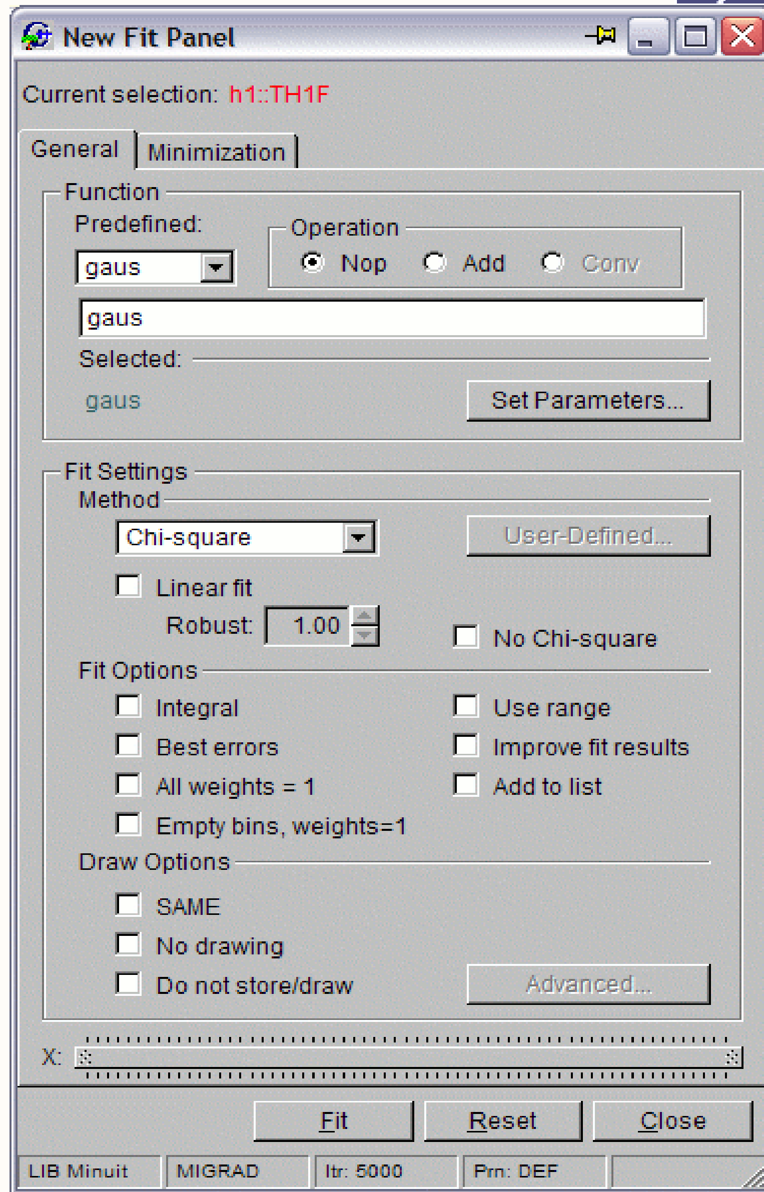
Fit Panel

To fit a histogram:

right click histogram,

"Fit Panel"

Straightforward interface
for fitting!



Parallel 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 ("nodes") run the analysis,
- collects and combines (merges) data,
- shows analysis results locally

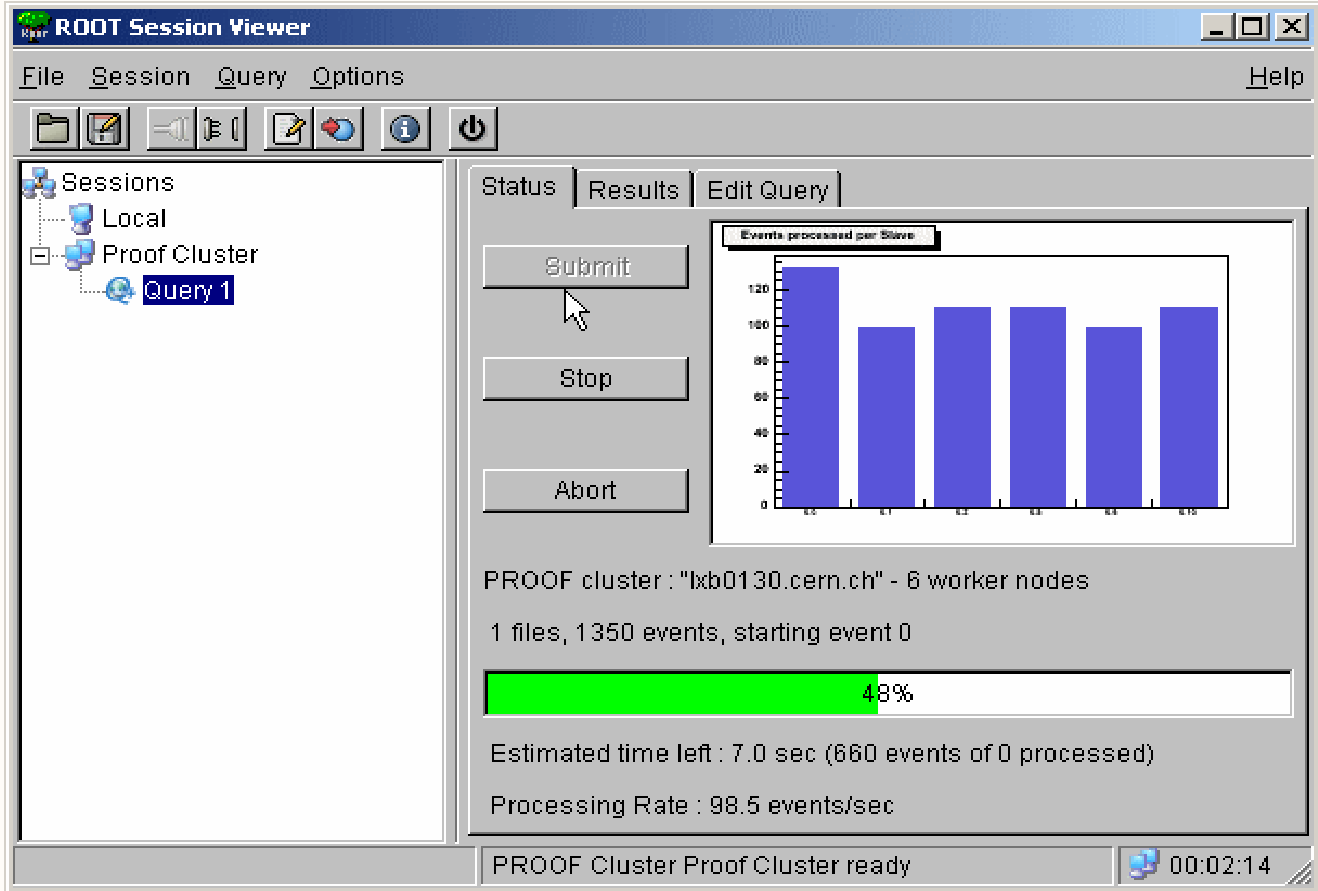
Including on-the-fly status reports!

A PROOF Session – Start

Starting a PROOF session is trivial:

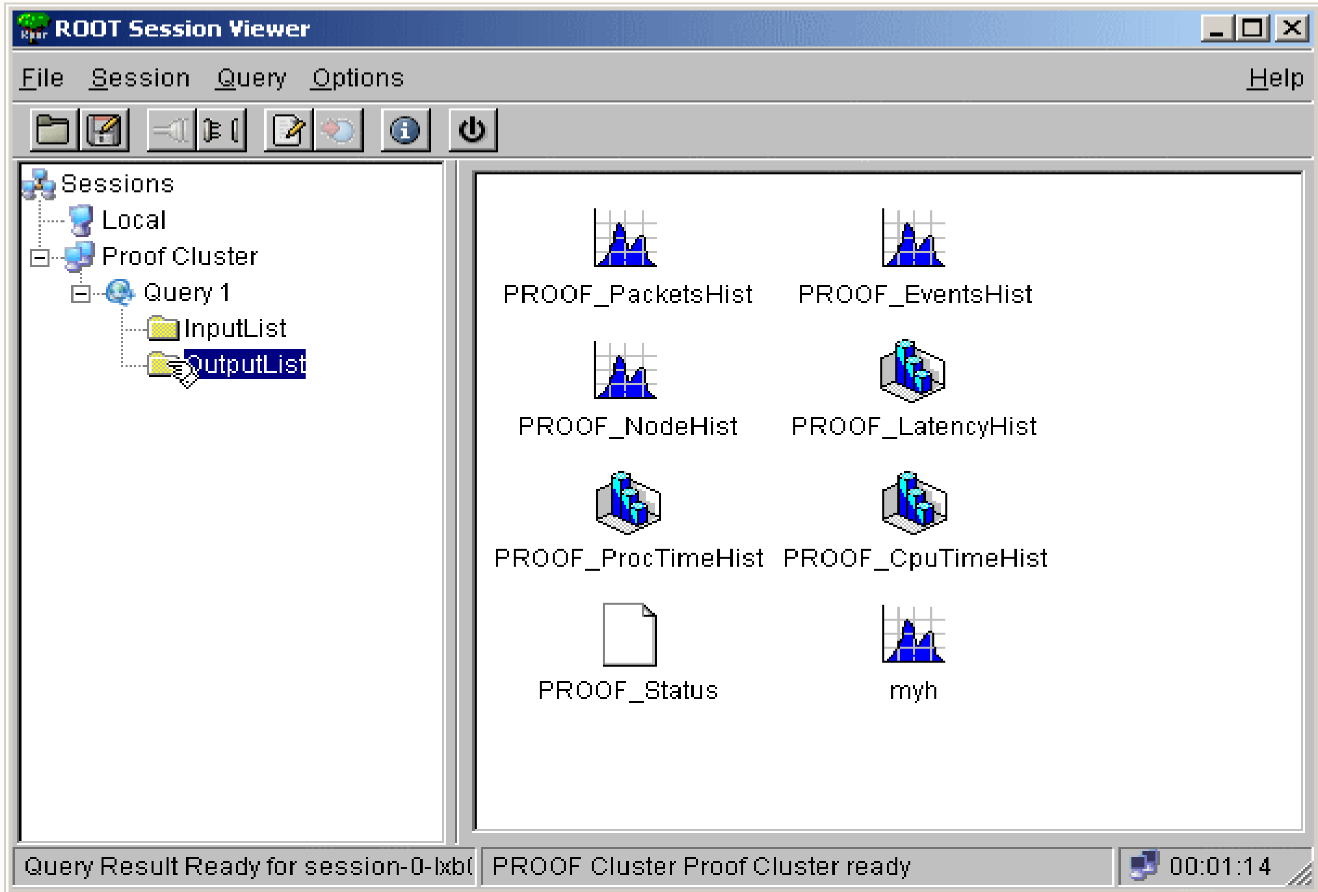
```
TProof::Open()
```

Opens GUI:



A PROOF Session – Results

Results accessible via TSessionViewer, too:



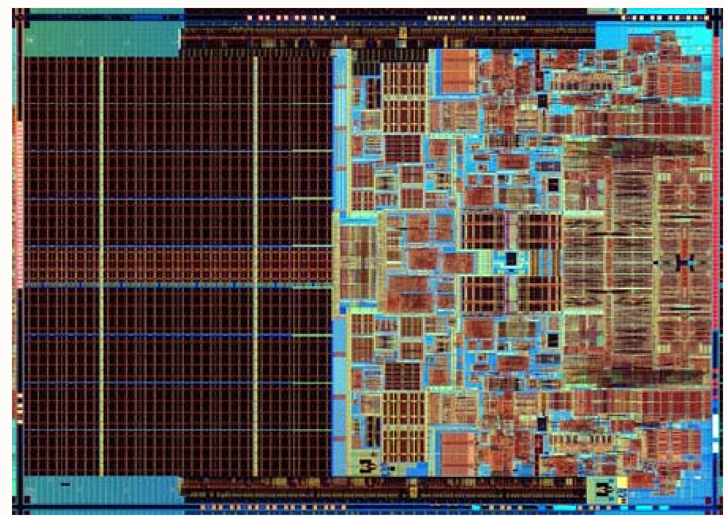
PROOF Documentation

Full sample session at

root.cern.ch/twiki/bin/view/ROOT/ProofGUI

But of course you need a little cluster of CPUs

Like your multicore laptop!



Summary

You've learned:

- analyzing a TTree can be easy and efficient
- integral part of physics is counting
- ROOT provides histogramming and fitting
- > 1 CPU: use PROOF!

Looking forward to hearing from you:

- as a user (help! bug! suggestion!)
- and as a developer!