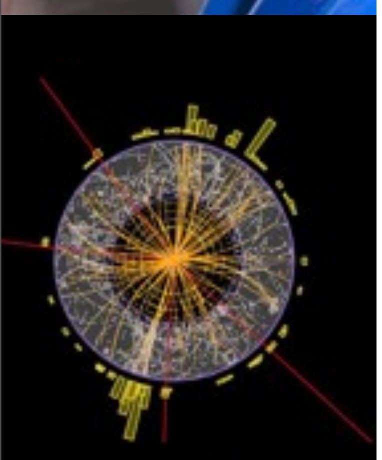
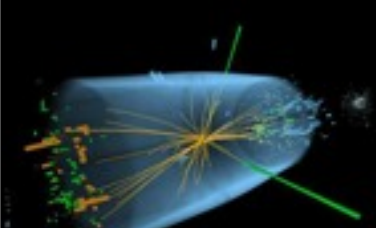




Introduction to ROOT I/O and Trees

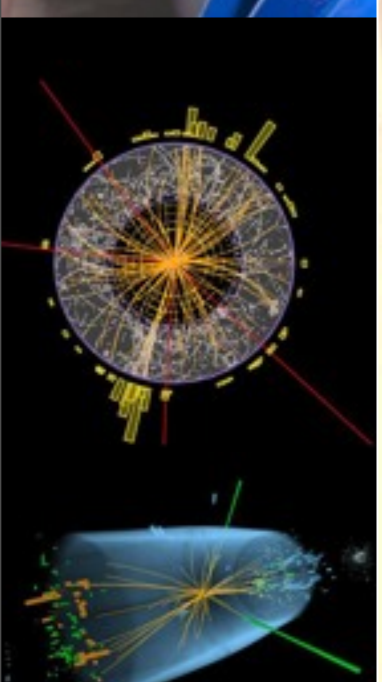
ROOT Training at IRMM

27th February 2013

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

Outline

- Introduction to I/O in ROOT
  - how to save ROOT objects in a file
    - reflection
  - example: saving an histogram
- ROOT Trees:
  - `TNtuple` class (a simple Tree)
  - `TTree` class
- How to create a Tree and to write in a file
- How to read and analyze the Tree
- Merging of Trees: `TChain`
- Using Tree Friends



Saving Data

Streaming, Reflection, TFile,
Schema Evolution

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

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);
```

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

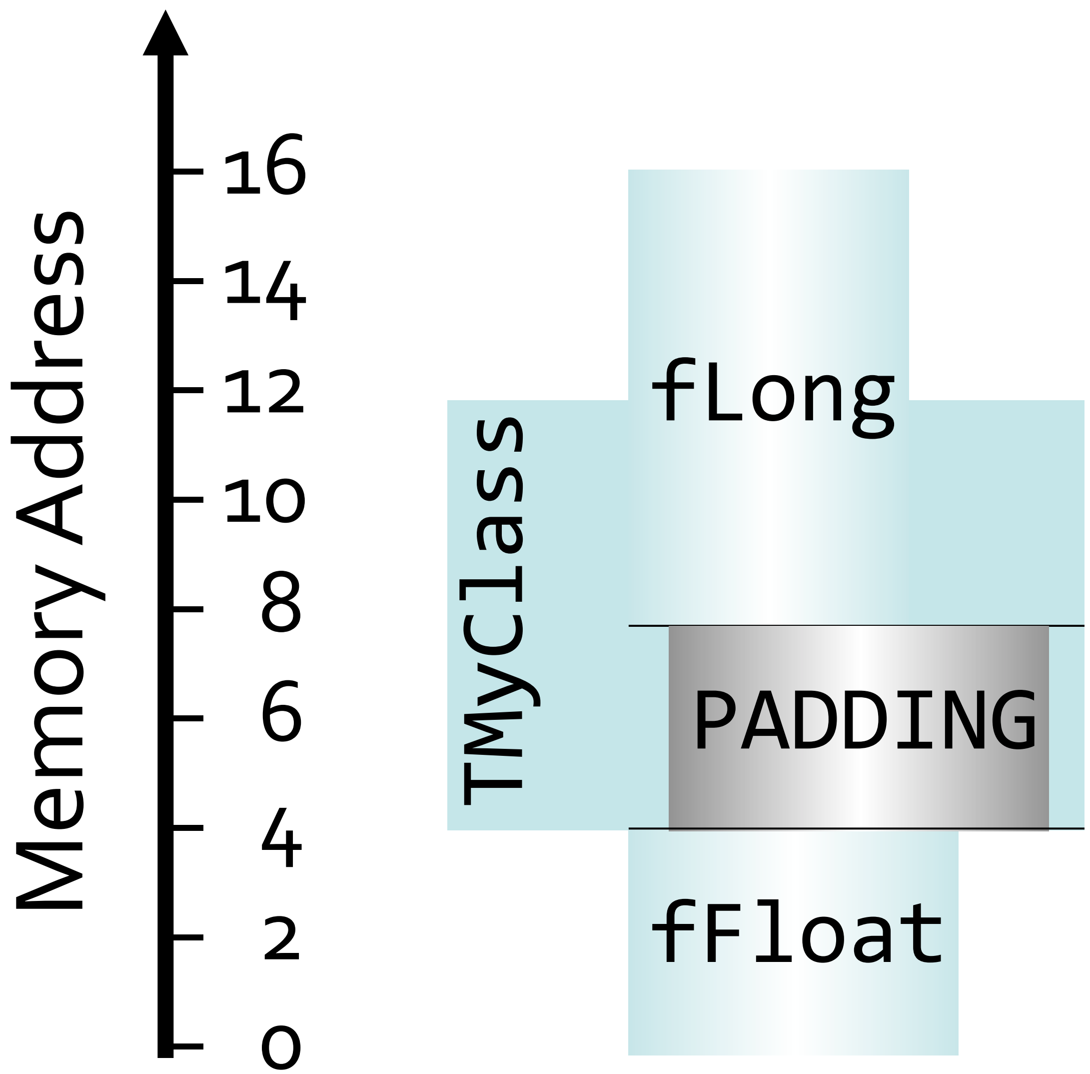
Signed	Unsigned	sizeof [bytes]
`Char_t`	`UChar_t`	1
`Short_t`	`UShort_t`	2
`Int_t`	`UInt_t`	4
`Long64_t`	`ULong64_t`	8
`Double32_t`		float on disk, double in RAM

Reflection

Need type description (platform dependent)

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

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

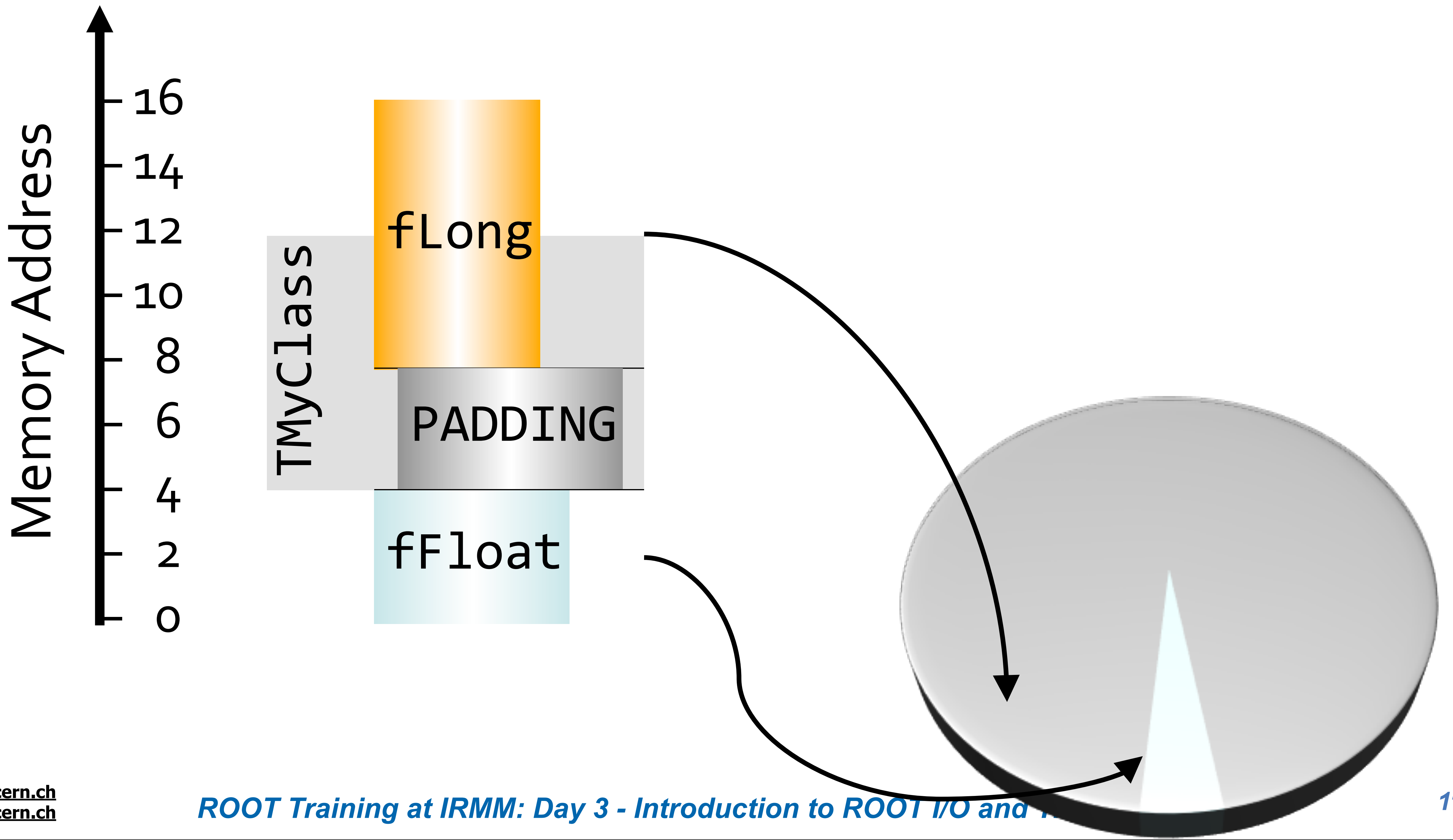


"fFloat" is at offset 0

"fLong" is at offset 8

I/O Using Reflection

members → memory → disk



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 in MyCode.cxx!

Dictionary needed for interpreter, too

ROOT has dictionary for all its types

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)1.685546875000000e+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);`

Reading a File

- Reading is simple:

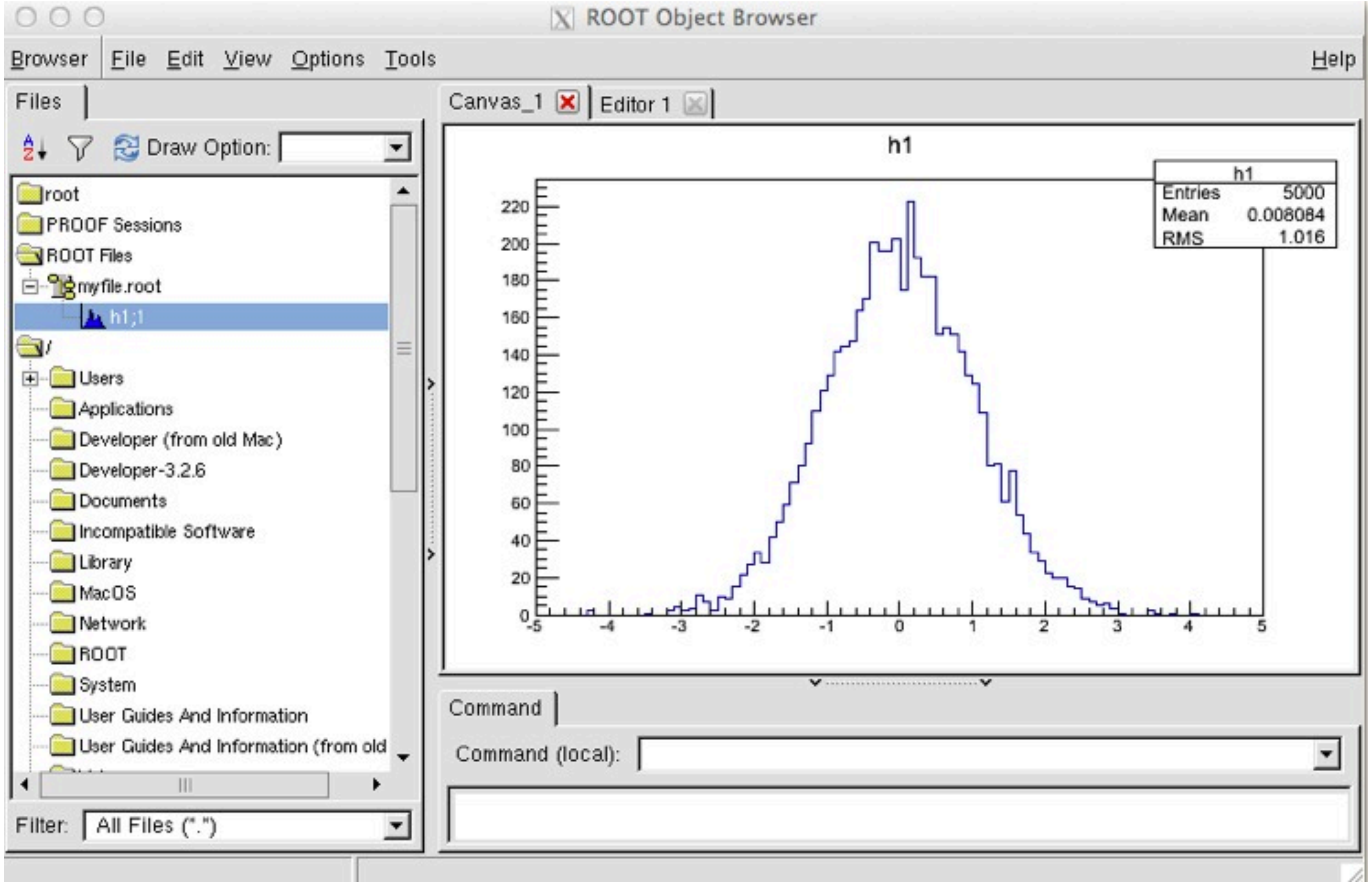
```
TFile* f = TFile::Open("myfile.root");
TH1* h = 0;
f->GetObject("h", h);
h->Draw();
delete f;
```

- 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`

TBrowser

- GUI for browsing ROOT objects written in a file

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



Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions here

https://twiki.cern.ch/twiki/bin/view/Main/RootIRMMTutorial2013IOandTreesExercises

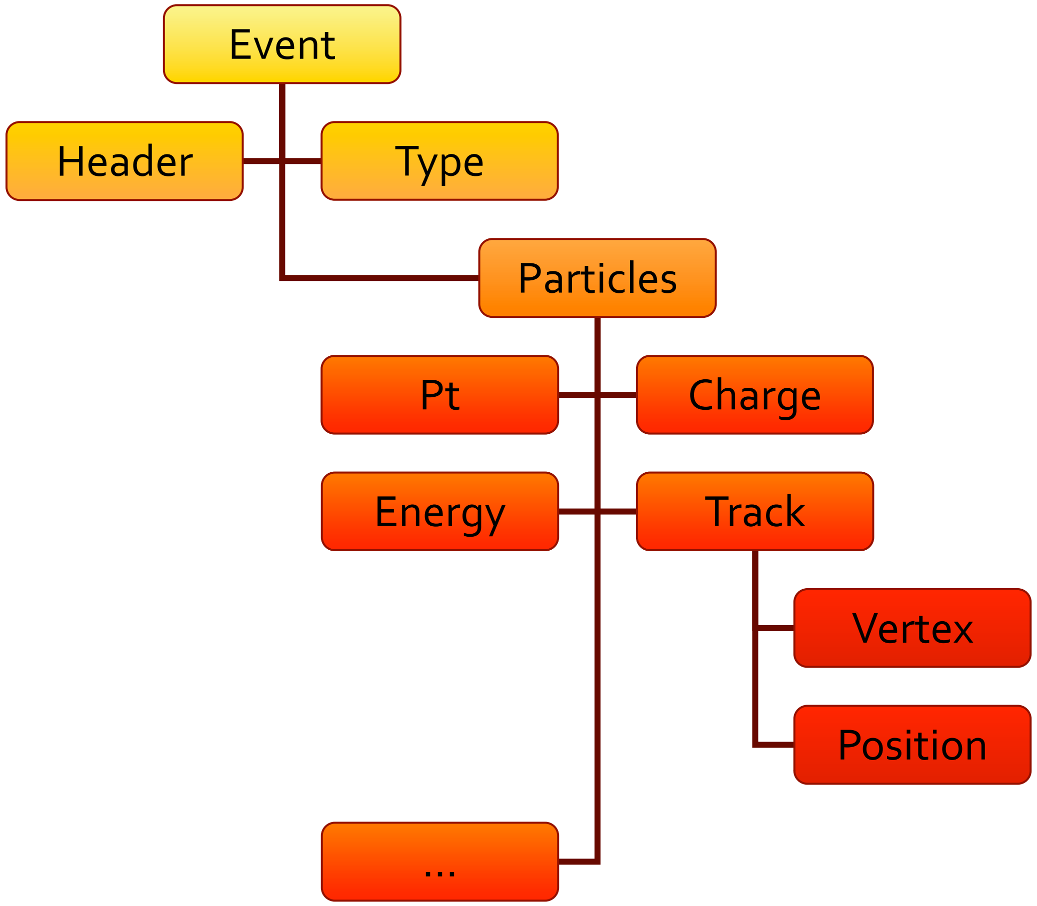
and solve exercise 1

Ntuple and Trees

- Ntuple class:
  - `TNtuple`
    - for storing tabular data
    - e.g. Excel Table with numbers

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

- Tree class
  - `TTree`
    - for storing complex data types
    - e.g. DataBase tables



Building ROOT Ntuple

- Creating and Storing N-tuples
  - The ROOT class `TNtuple` can store only floating entries
    - each raw (record) must be composed only of floating types
  - Specify the name (label) of the type when creating the object

```
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();
```

How To Read a NTuple

- Open the file and get the ntuple object

```
TFile f("ntuple_data.root");
ntuple->Print();
```

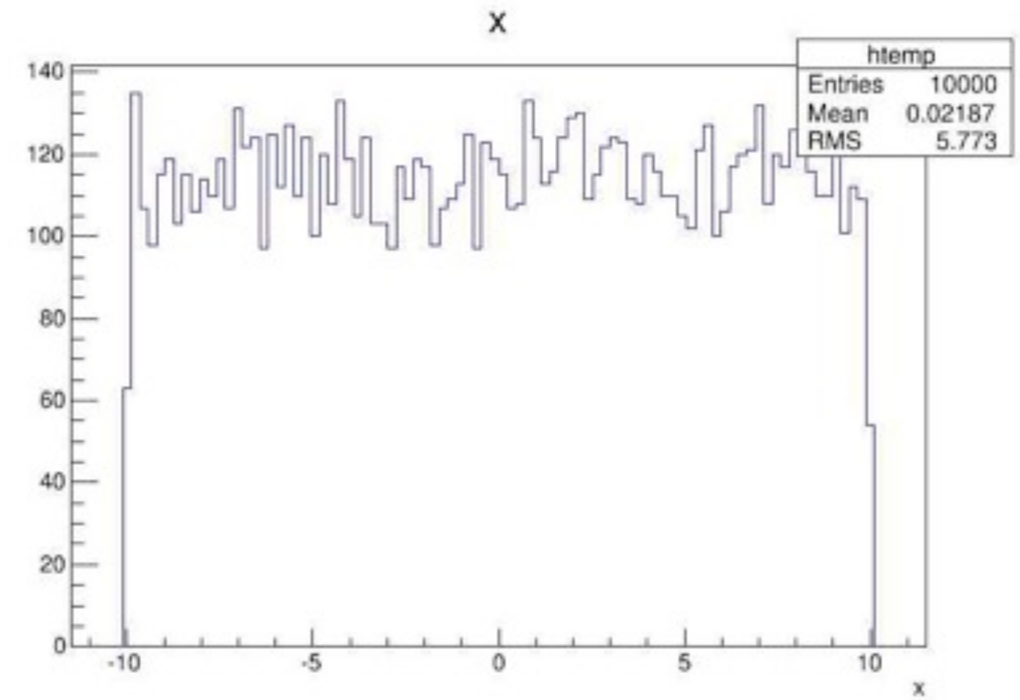
Note that (as for histograms) we do not need to use TFile::Get
This works only in CINT, not valid C++

```
******************************************************************************
*Tree    :ntuple    : Example N-tuple                                        *
*Entries :    10000 : Total =          290753 bytes  File  Size =     161076 *
*        :          : Tree compression factor =   1.80                       *
******************************************************************************
*Br    0 :x         : Float_t                                                *
*Entries :    10000 : Total  Size=      72596 bytes  File Size  =      40140 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   1.80     *
*............................................................................*
*Br    1 :y         : Float_t                                                *
*Entries :    10000 : Total  Size=      72596 bytes  File Size  =      40140 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   1.80     *
*............................................................................*
*Br    2 :z         : Float_t                                                *
*Entries :    10000 : Total  Size=      72596 bytes  File Size  =      40140 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   1.80     *
*............................................................................*
*Br    3 :t         : Float_t                                                *
*Entries :    10000 : Total  Size=      72596 bytes  File Size  =      40140 *
*Baskets :        2 : Basket Size=      32000 bytes  Compression=   1.80     *
*............................................................................*
```

Looking at the Ntuple

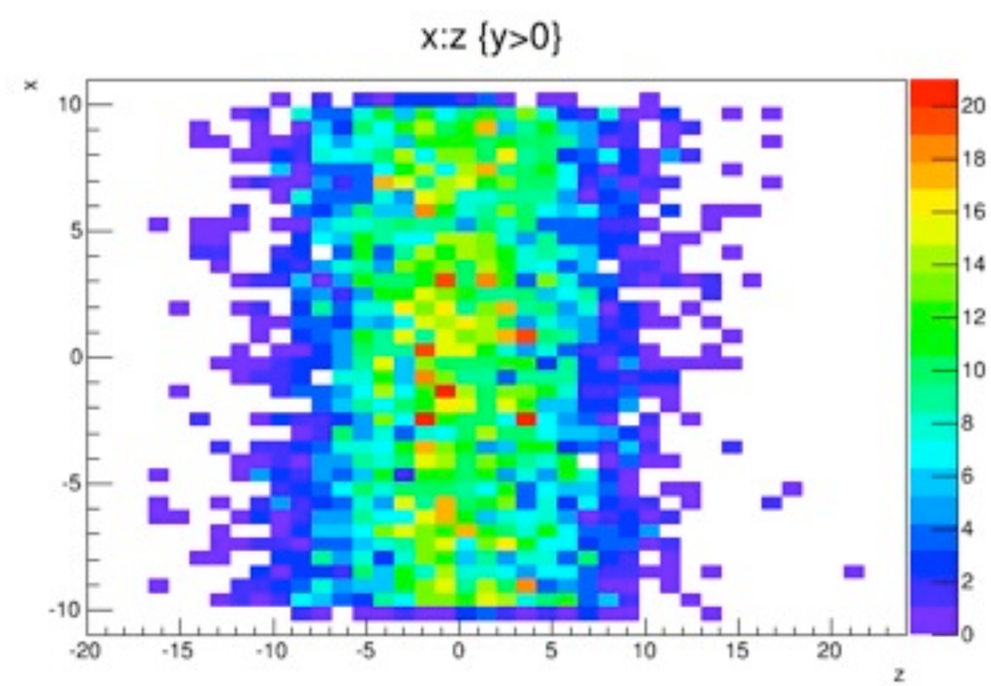
- Can Draw one of the variable of the ntuple:

```
ntuple->Draw("x")
```



- Can Draw 2 (or more) variables:

```
ntuple->Draw("x:z","y>0","colz")
```



- Can Scan the variables' values:

```
ntuple->Scan("x:y:z:t")
```

```
************************************************************
*    Row   *         x *         y *         z *         t *
************************************************************
*        0 * 9.9948349 * -6.741802 * 5.5279893 * 14.624994 *
*        1 * -0.300527 * 9.1495389 * -0.283586 * 6.1610527 *
*        2 * 4.7990598 * 5.1988759 * -2.053815 * 11.531604 *
*        3 * 6.0880603 * 0.3934423 * -0.244702 * 9.3569278 *
*        4 * -5.566646 * -5.736190 * -7.190714 * 10.979947 *
*        5 * -6.117022 * 8.8743352 * -3.680147 * 1.0724577 *
*        6 * 3.3112785 * -0.027793 * -4.079026 * 17.021858 *
*        7 * -4.069489 * -7.651821 * 5.8266844 * 4.3367080 *
*        8 * 4.5083704 * 2.7426230 * -0.912180 * 23.068317 *
*        9 * 3.9853439 * -7.843750 * 0.0240303 * 15.712759 *
*       10 * -4.221793 * -8.336503 * 0.4737141 * 24.971460 *
*       11 * -4.157171 * 7.8324747 * -0.681545 * 19.608907 *
```

Getting The Entries

- Entries of a ROOT N-tuple can be retrieved using `TNtuple::GetEntry(irow)`

```
TFile f("ntuple_data.root");

TNtuple *ntuple=0;
f.GetObject("ntuple",ntuple);

// loop on the ntuple entries
for (int i = 0; i < ntuple->GetEntries(); ++i) {

   ntuple->GetEntry(i);
   float * raw_content = ntuple->GetArgs();
   float x = raw_content[0];
   float y = raw_content[1];
   float z = raw_content[2];
   float t = raw_content[3];

   // do something with the data..

}
```

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions here

https://twiki.cern.ch/twiki/bin/view/Main/RootIRMMTutorial2013IOandTreesExercises

and solve exercise 2

ROOT Trees

- ROOT N-tuple can store only floating point variables
- For storing complex types, i.e. objects we can use the ROOT tree class, TTree
  - TNtuple is a special case of a TTree (a derived class)

- The ROOT Tree is
  - extremely efficient write once, read many.
  - Designed to store >109 (HEP events) with same data structure.
  - Trees allow fast direct and random access to any entry (sequential access is the best).
  - Optimized for network access (read-ahead).

Why Trees ?

- `object.Write()` is convenient for simple objects like histograms, but inappropriate for saving collections of events containing complex objects
- Reading a collection:
  - read all elements (all events)
- With trees:
  - only one element in memory,
  - or even only a part of it (less I/O)
- Trees buffered to disk (TFile);
  - I/O is integral part of TTree concept

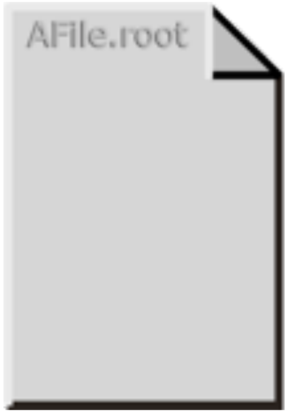
Tree Access

- Databases have row wise access
  - Can only access the full object (e.g. full event)
- ROOT trees have column wise access
  - Direct access to any event, any branch or any leaf even in the case of variable length structures
  - Designed to access only a subset of the object attributes (e.g. only particles' energy)
  - 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!

Building a ROOT Tree

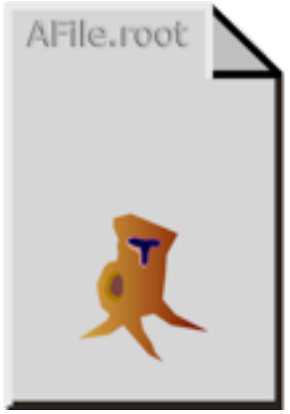
- Steps to build a Tree
  - Create a TFile class
    - Tree can be huge → need file for swapping filled entries

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



  - Create a TTree class

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



  - Add a Branch (TBranch) to the TTree
  - Fill the tree with the data
  - Write the tree to file

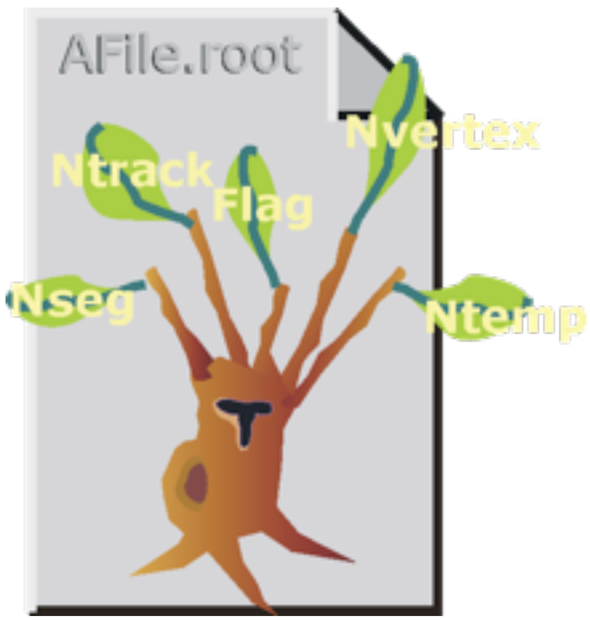
Tree Structure and Branches

- What is a Branch ?
  - A branch is like a directory
    - it can hold a simple variable, a list of variables, an object or evan a collection of objects
    - The leaves 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
    - branches of the same tree can be written to separate files

Adding a Branch to the Tree

- To add a branch we need:
  - Name of the Branch
  - Address of the pointer to the object we want to store

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



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

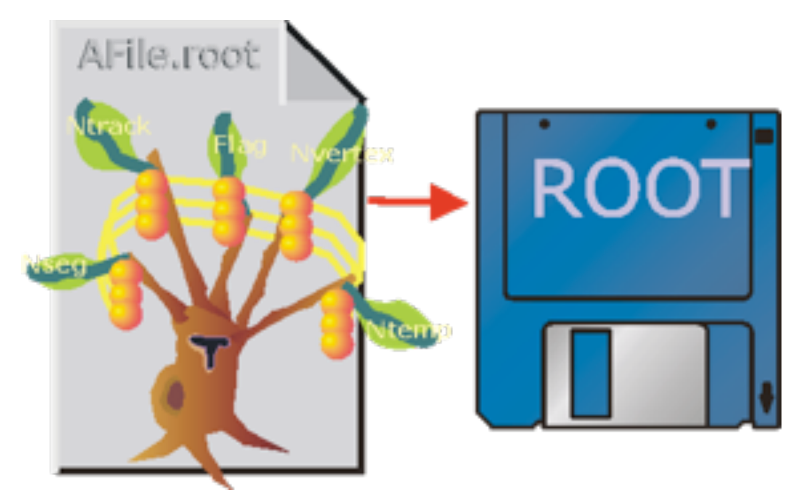
Fill the Tree

- Loop on the tree
- assign values to the object we want to store
- 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 hypothetical 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("EventBranch", &myEvent);
   for (int e=0;e<100000;++e) {
      myEvent->Generate();  // hypothetical
      t->Fill();
   }
   t->Write();
}
```

Note: Event is an hypothetical class provided by the user

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions here

https://twiki.cern.ch/twiki/bin/view/Main/RootIRMMTutorial2013IOandTreesExercises

and solve exercise 3

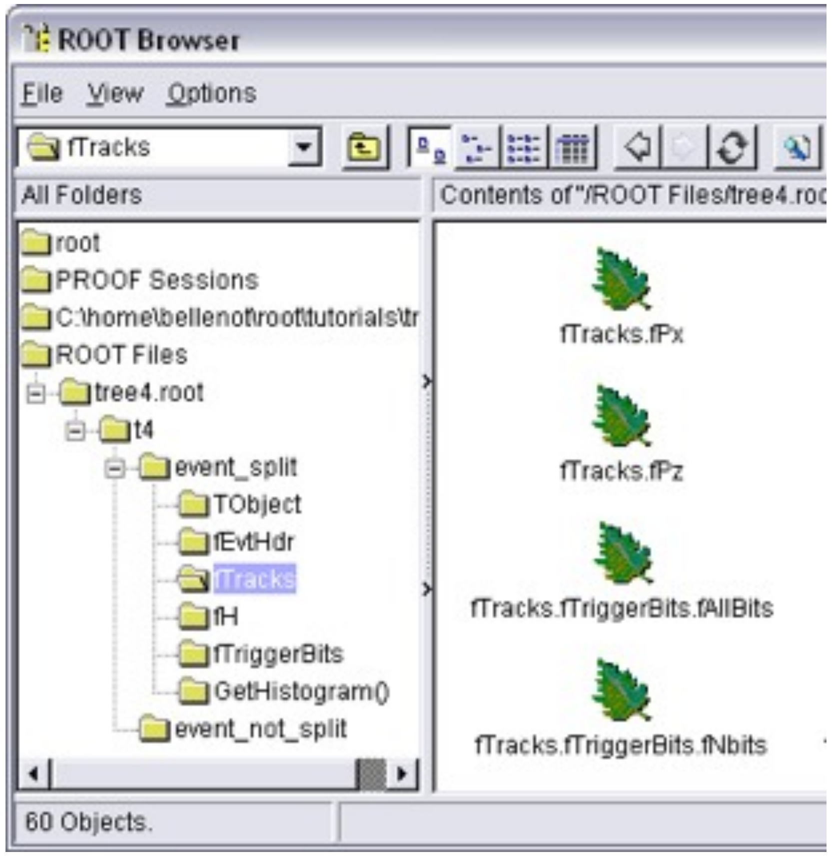
Reading a Tree

- Open the file and get the TTree object from the file
  - same as we have seen for TNtuple

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

- Or browse the TTree using the TBrowser

- TTree::Print() shows the data layout



How To Read a Tree

- Create a variable pointing to the data

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

- Associate a branch with the variable

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

- Read ith-entry in the Tree

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

```
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 *T = (TTree*)f->Get("T");
  Event *myEvent = 0;
  TBranch* brEvent = 0;
  T->SetBranchAddress("EvBranch", &myEvent, brEvent);
  T->SetCacheSize(10000000);
  T->AddBranchToCache("EvBranch");
  Long64_t nent = T->GetEntries();
  for (Long64_t i = 0; i < nbent; ++i) {
     brEvent->GetEntry(i);
     myEvent->Analyze();
  }
```

Data pointers (e.g. myEvent) MUST be set to 0

Accessing Tree Branches

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

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

Tree Selection Syntax

- 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 for `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,…
```

Time for Exercises!

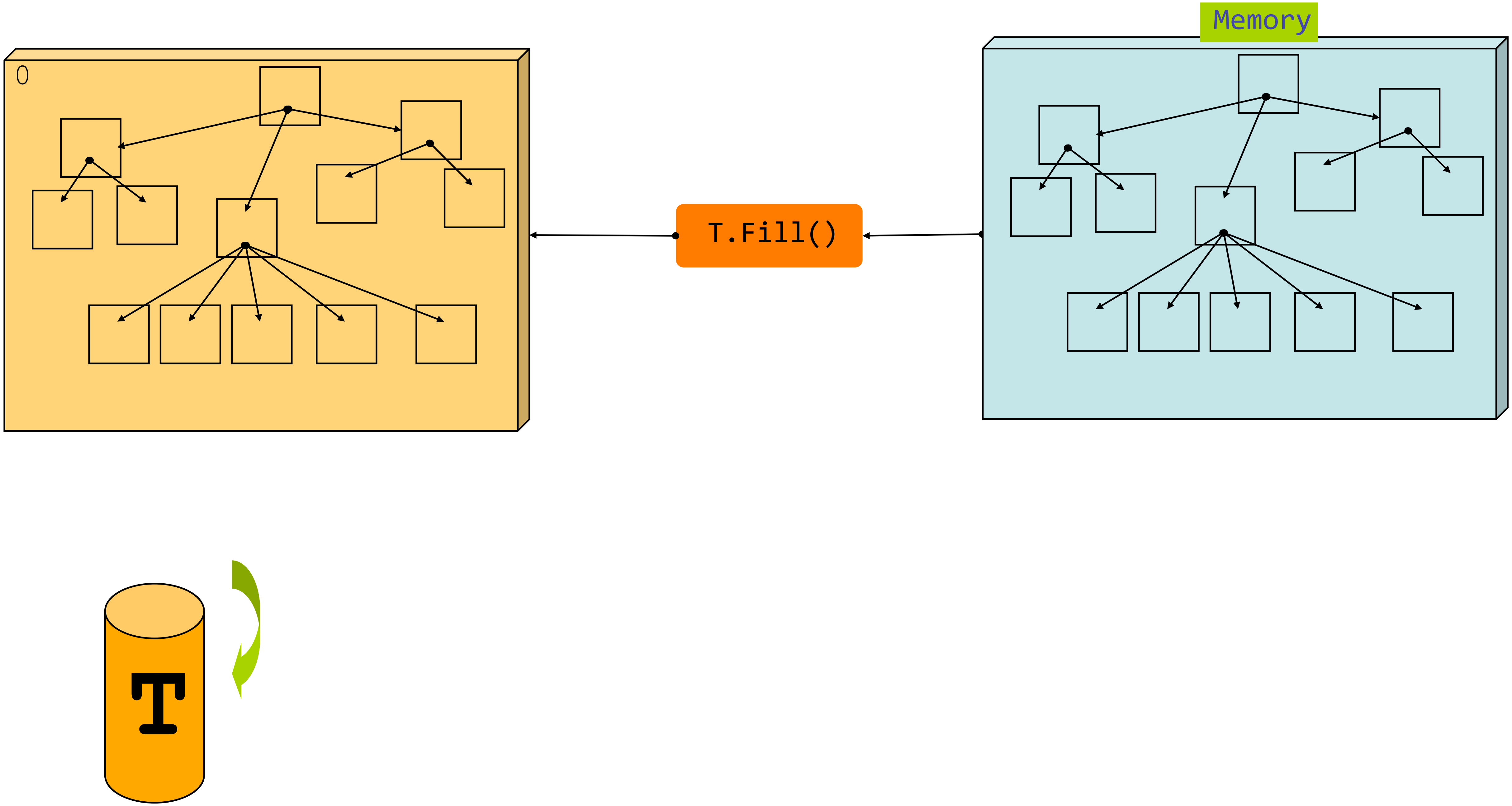
Put in practice the concepts to which you were just exposed: read the instructions here

https://twiki.cern.ch/twiki/bin/view/Main/RootIRMMTutorial2013IOandTreesExercises

and solve exercise 4

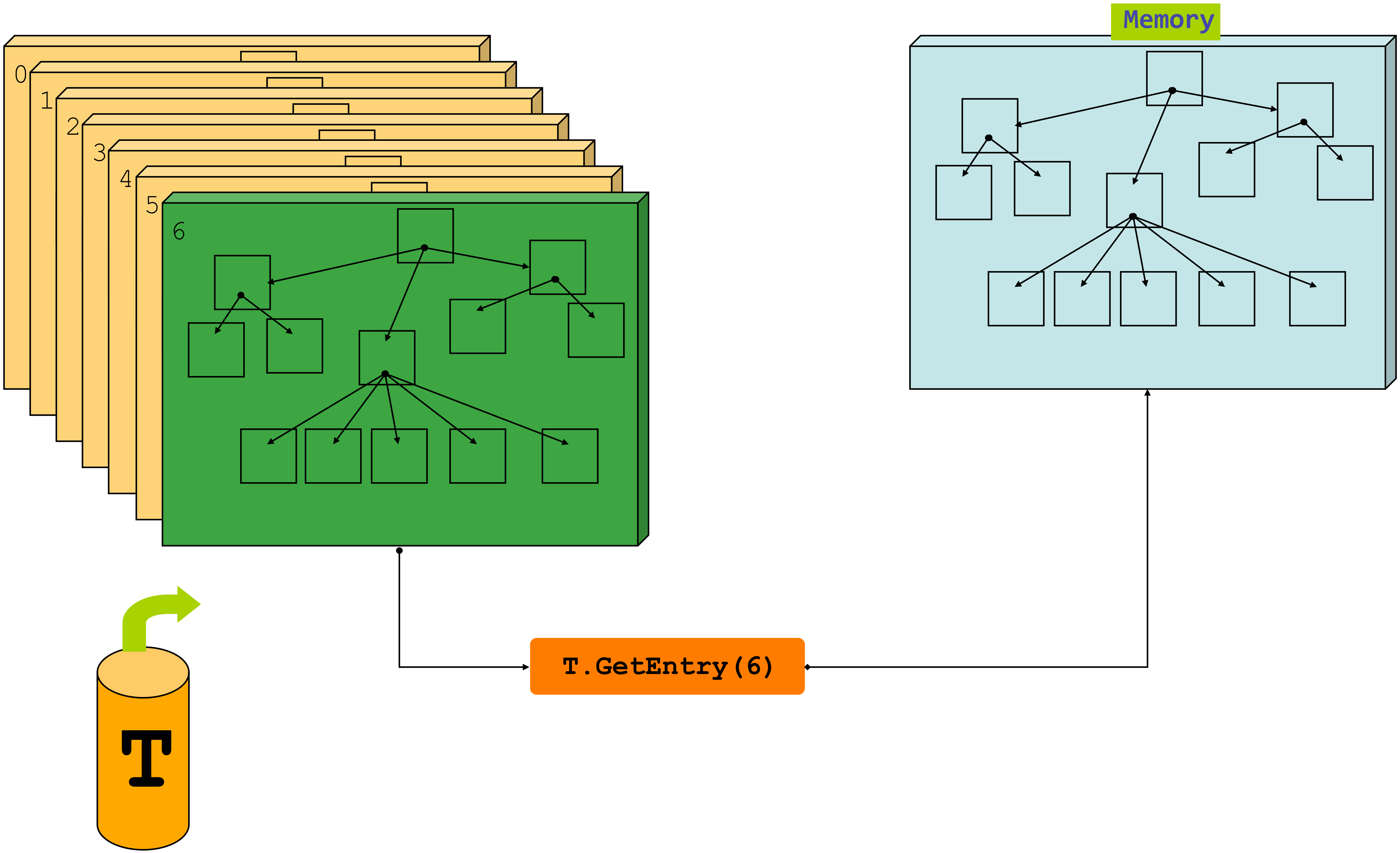
Memory ↔ Tree

- Each Node is a branch in the Tree



Memory ↔ Tree

- Each Node is a branch in the Tree



TChain: The Forest

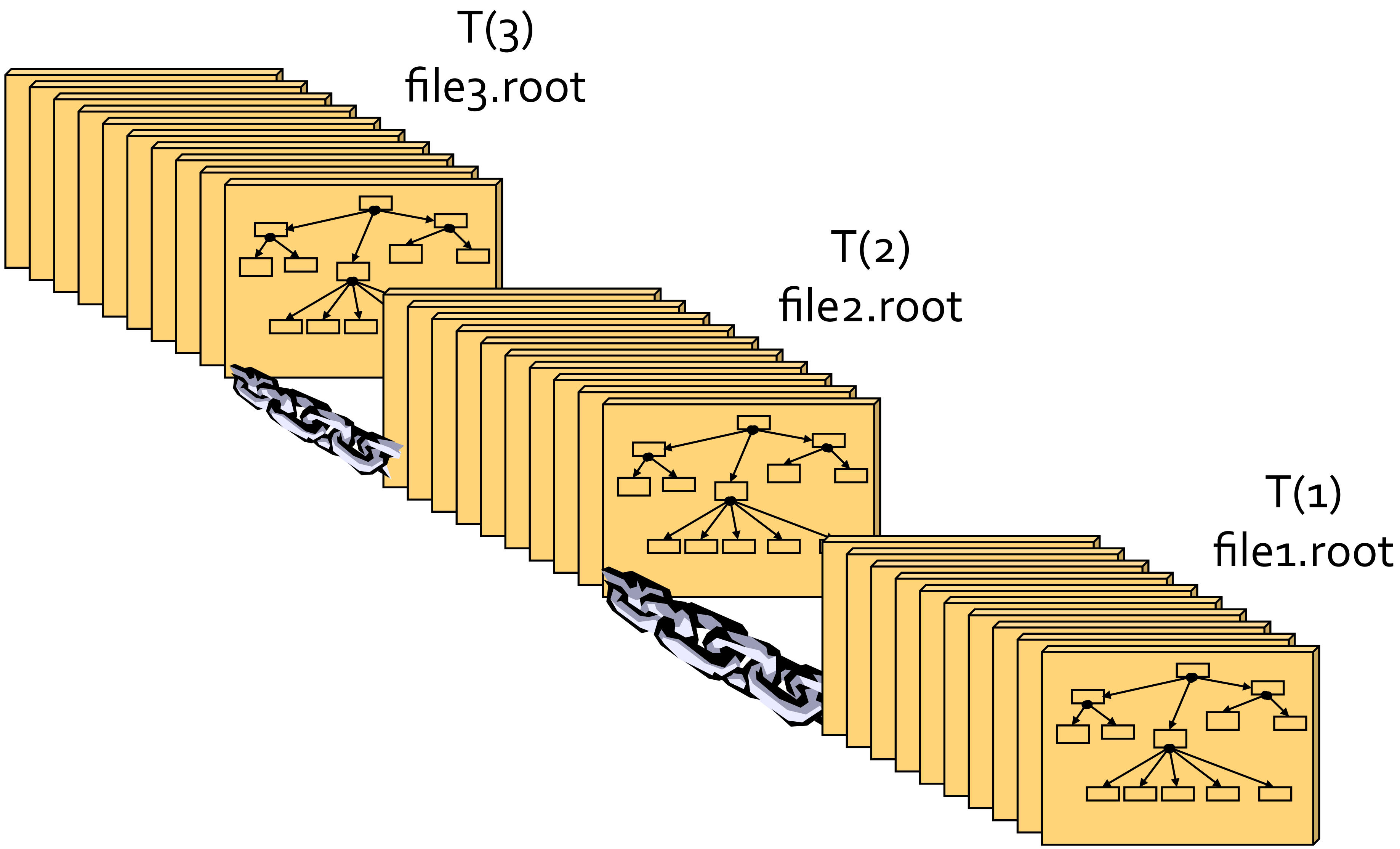
- Collection of Trees:
  - list of ROOT files containing the same tree
- Same semantics as `TTree`.
  - As an example, assume we have three files called file1.root, file2.root, file3.root. Each contains tree called "T". Create a chain:

```
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!`

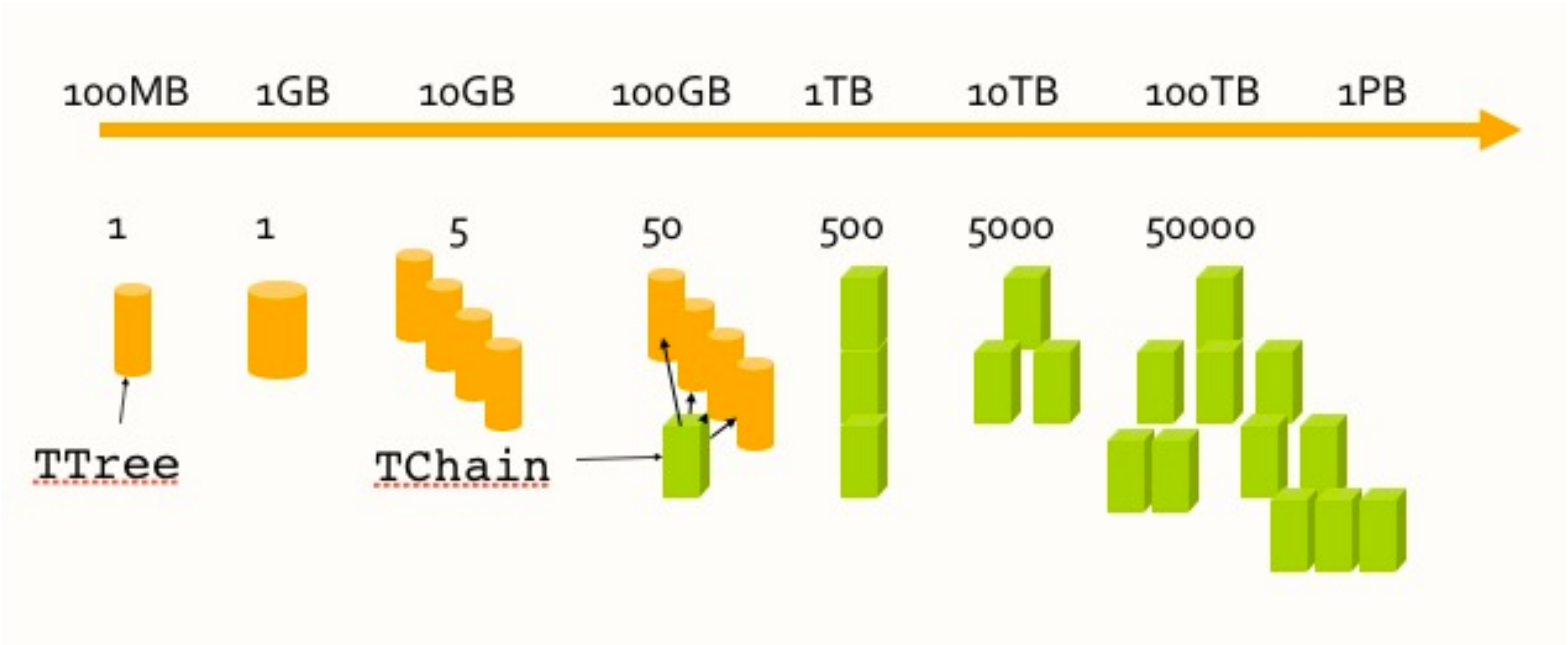
TChain

- Chain Files together



Data Volume and Organization

- A `TFile` typically contains 1 `TTree`
- A `TChain` is a collection of `TTrees` or/and `TChains`

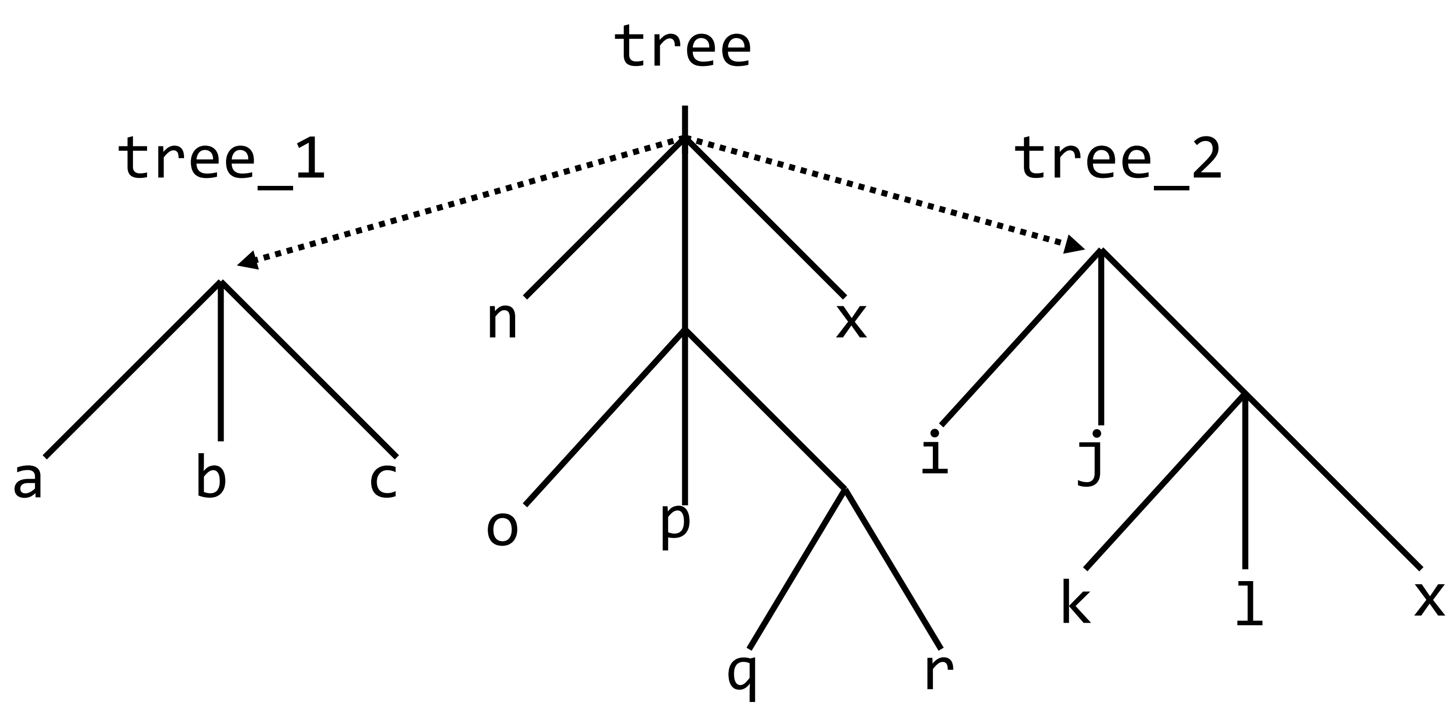


Tree Friends

- 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");
```

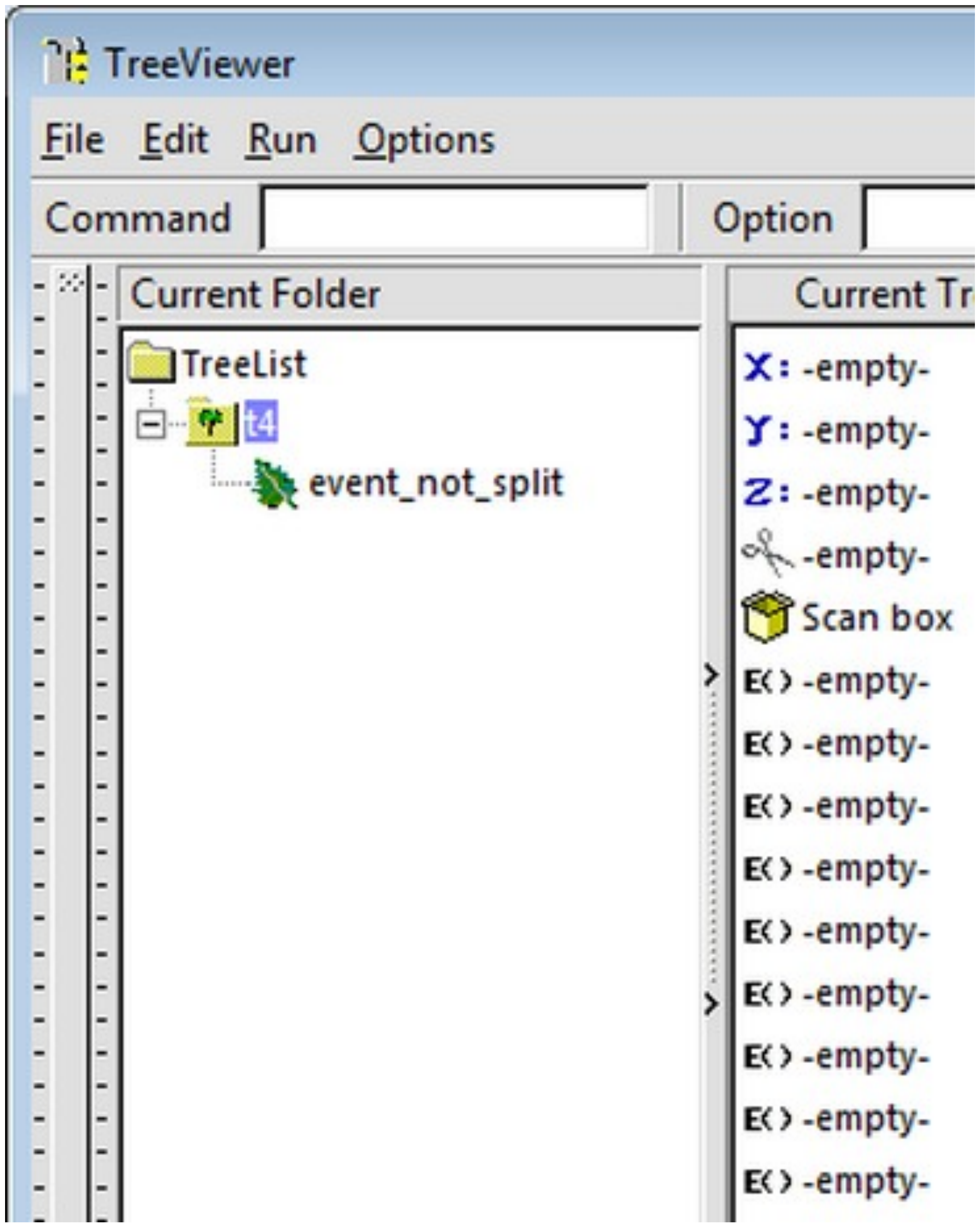
Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions here

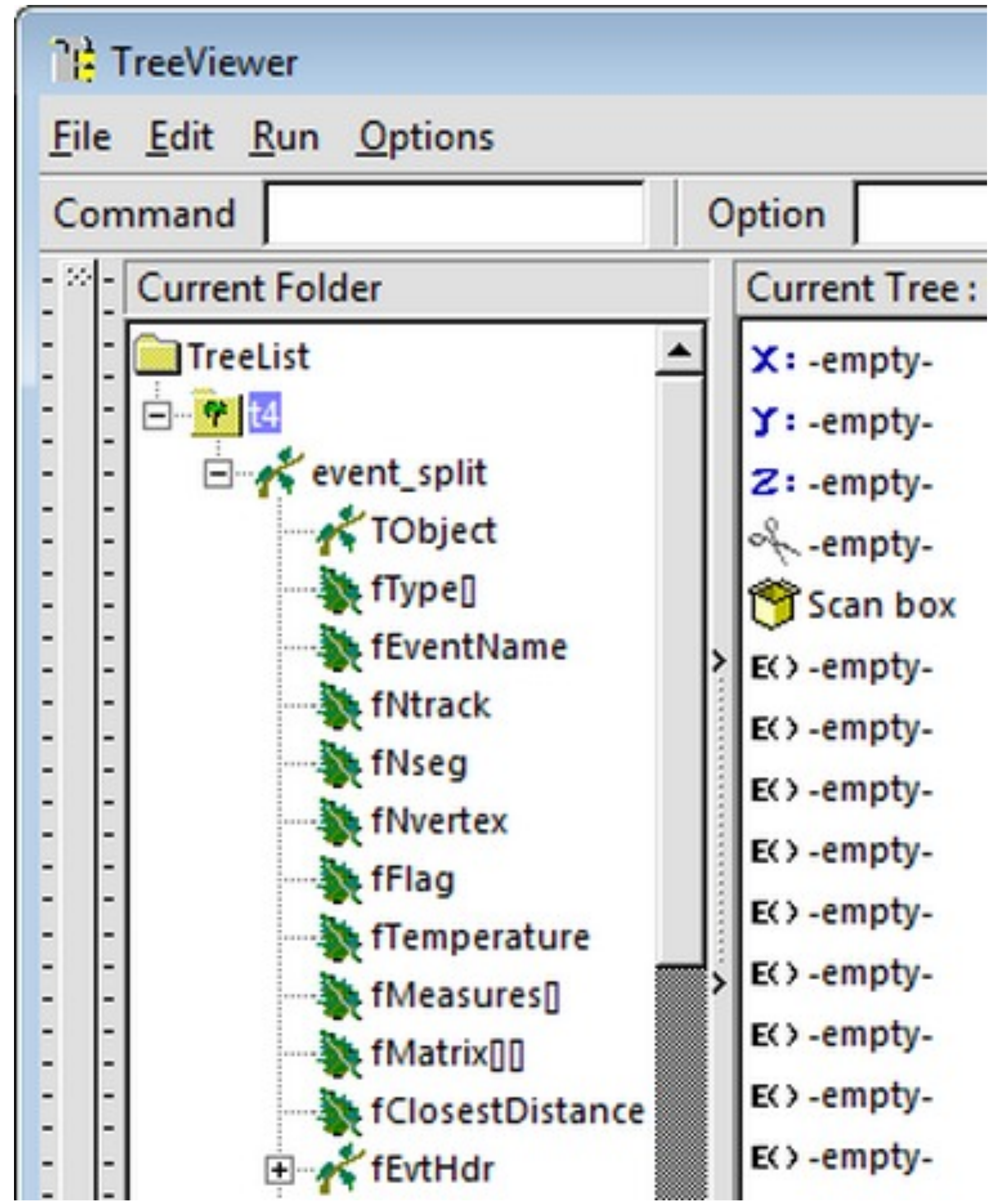
https://twiki.cern.ch/twiki/bin/view/Main/RootIRMMTutorial2013IOandTreesExercises

and solve exercise 5 and 6

Splitting



Split level = 0



Split level = 99

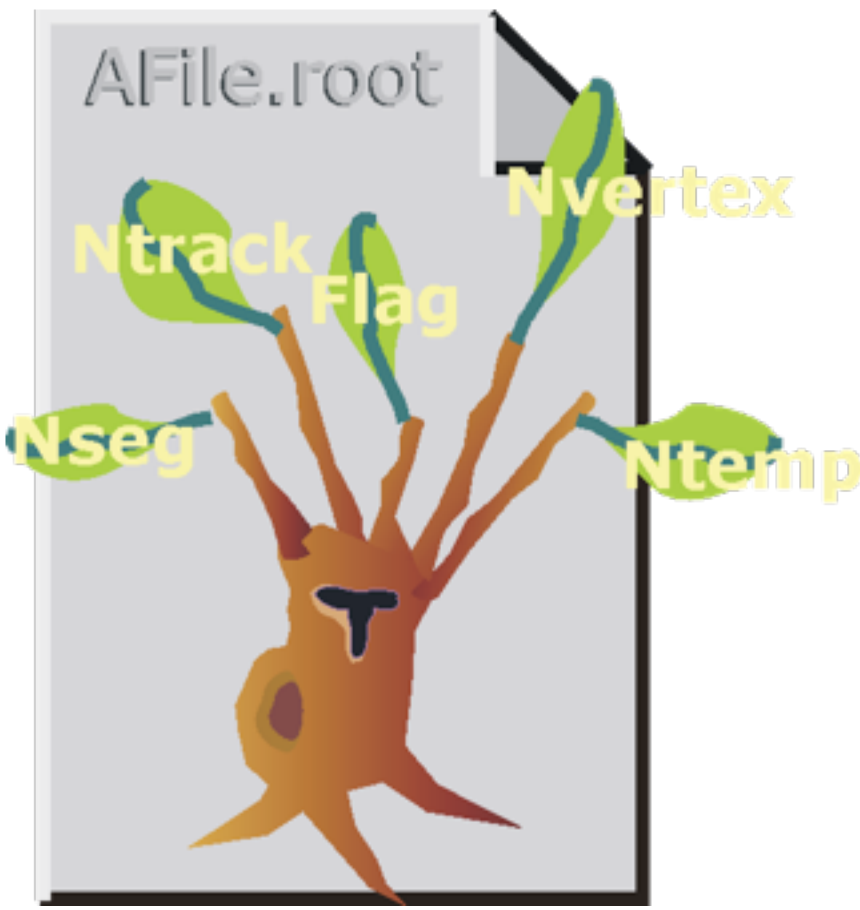
Splitting

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

Splitting

Setting the split level (default = 99)

Split level = 0



Split level = 99

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

Performance Considerations

A split branch is:

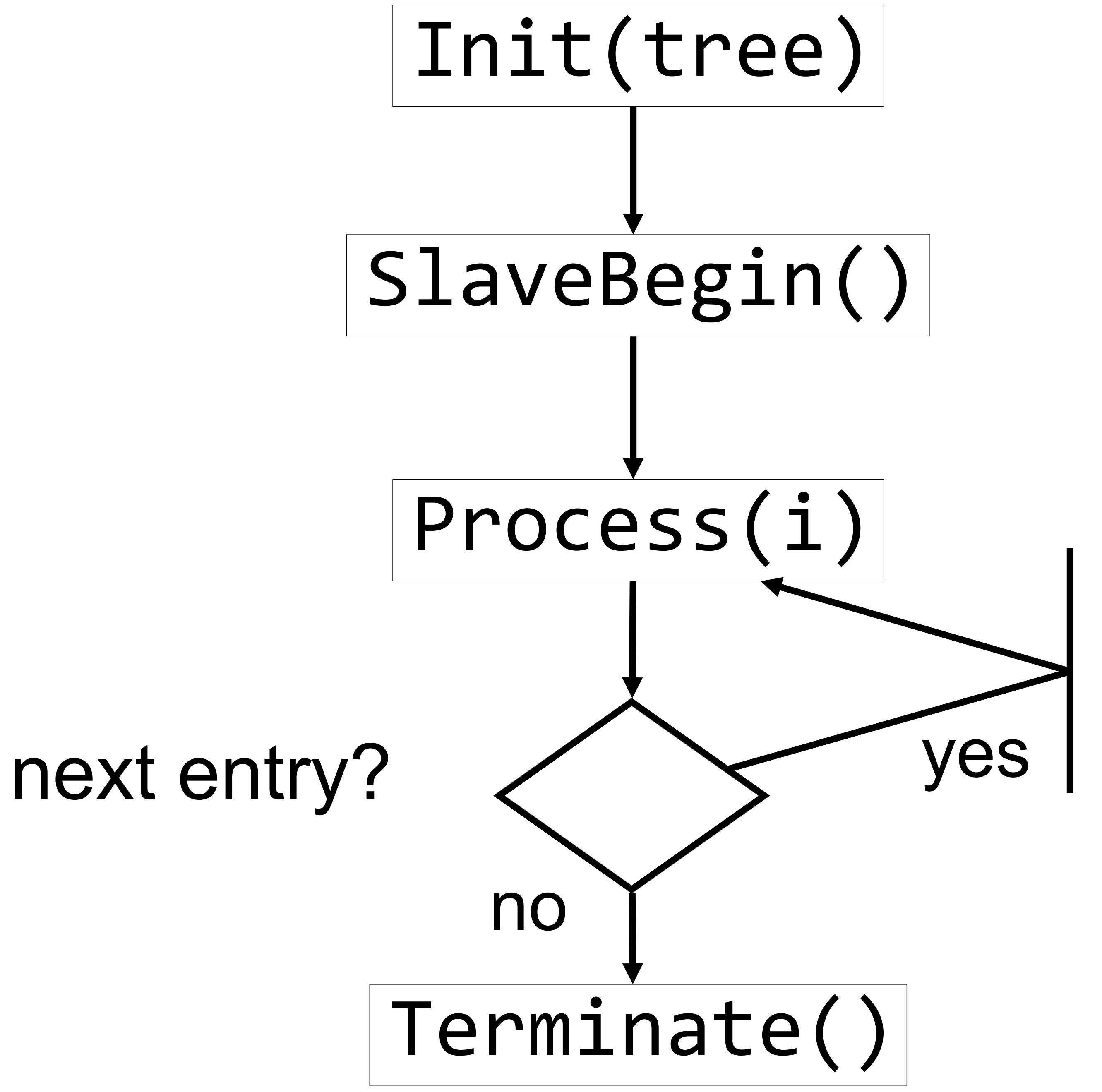
- Faster to read – if you only want a subset of data members
- Slower to write due to the large number of branches

Analyzing Trees

- 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.

Tree Data Access

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



TSelector

Steps of ROOT using a `TSelector`:

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

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,…

Time for Exercises!

Put in practice the concepts to which you were just exposed: read the instructions here

https://twiki.cern.ch/twiki/bin/view/Main/RootIRMMTutorial2013IOandTreesExercises

and solve exercise 6

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