



Trees: New Developments Folders and Tasks

ROOT Workshop 2001
June 13 FNAL

René Brun

<http://root.cern.ch>



Trees before 3.01

- TTree
- TBranch
- TBranchObject
- TBranchClones
- TChain

Many limitations in split mode

Only simple classes in TClonesArray

Original classes required for TBranchObject



Trees in version 3.01

- TTree
- TBranch
- TBranchObject
- TBranchClones
- TBranchElement ←
- TChain
- TFriendElement ←

This new class replaces
TBranchObject & TBranchClones
Many more cases supported

Tree Friends
A fundamental addition



Self-describing files

- Dictionary for persistent classes written to the file.
- ROOT files can be read by foreign readers (JAS)
- Support for Backward and Forward compatibility
- Files created in 2001 must be readable in 2015
- Classes (data objects) for all objects in a file can be regenerated via `TFile::MakeProject`

```
Root > TFile f("demo.root");
```

```
Root > f.MakeProject("dir", "*", "new++");
```

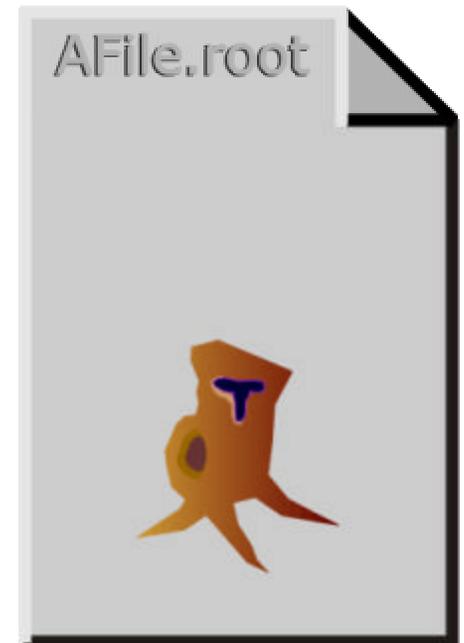
Create a TTree Object



A tree is a list of branches.

The TTree Constructor:

- Tree Name (e.g. "myTree")
- Tree Title

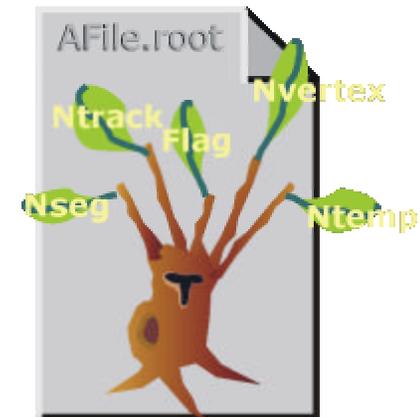


```
TTree *tree = new TTree("T", "A ROOT tree");  
TTree T2("T2", "/Event/Tracks");
```

Adding a Branch



- Branch name
- Class name
- Address of the pointer to the Object (descendant of TObject)
- Buffer size (default = 32,000)
- Split level (default = 1)



```
Event *event = new Event();  
myTree->Branch("eBranch", "Event", &event, 64000, 1);
```

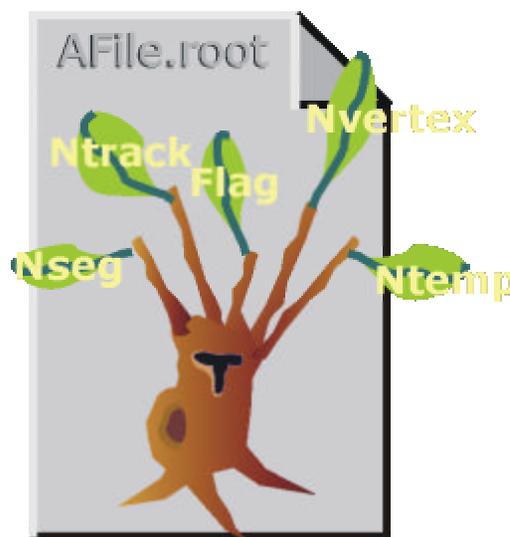


Splitting a Branch

Setting the split level (default = 1)



Split level = 0



Split level = 1

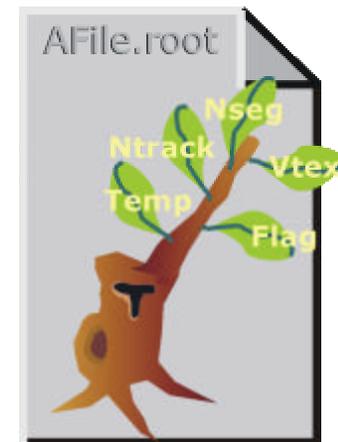
Example:

```
int splitlevel = 2;  
tree->Branch("EvBr", "Event", &ev, 64000, splitlevel);
```

Adding Branches with a List of Variables



- Branch name
- Address: the address of the first item of a structure.
- Leaflist: all variable names and types
- Order the variables according to their size



Example

```
TBranch *b = tree->Branch ("Ev_Branch", &event,  
"ntrack/I:nseg:nvtex:flag/i:temp/F");
```

Adding Branches with a TClonesArray



- Branch name
- Address of a pointer to a TClonesArray
- Buffer size
- Split level (default = 1)



Example:

```
tree->Branch( "tracks",&Track,64000,1);  
tree->Branch( "tracks","TClonesArray",&Track,64000,1);
```

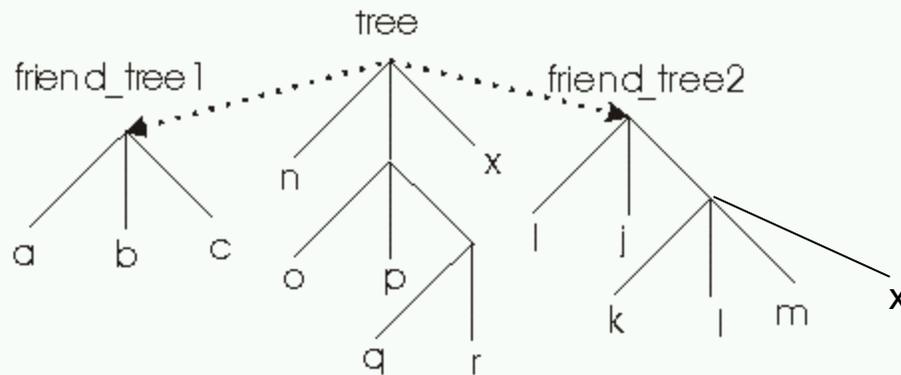


Chains of Trees

- A TChain is a collection of Trees.
- Same semantics for TChains and TTrees
 - `root > .x h1chain.C`
 - `root > chain.Process("h1analysis.C")`
 - `root > chain.Process("h1analysis.C+")`

```
void h1chain() { h1chain.C  
  //creates a TChain to be used by the h1analysis.C class  
  //the symbol H1 must point to a directory where the H1 data sets  
  //have been installed  
  
  TChain chain("h42");  
  chain.Add("$H1/dstarmb.root");  
  chain.Add("$H1/dstarp1a.root");  
  chain.Add("$H1/dstarp1b.root");  
  chain.Add("$H1/dstarp2.root");  
}
```

Tree Friends



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

```
Root > TFile f1("tree1.root");
```

```
Root > tree.AddFriend("tree2", "tree2.root")
```

```
Root > tree.AddFriend("tree3", "tree3.root");
```

```
Root > tree.Draw("x:a", "k<c");
```

```
Root > tree.Draw("x:tree2.x", "sqrt(p)<b");
```



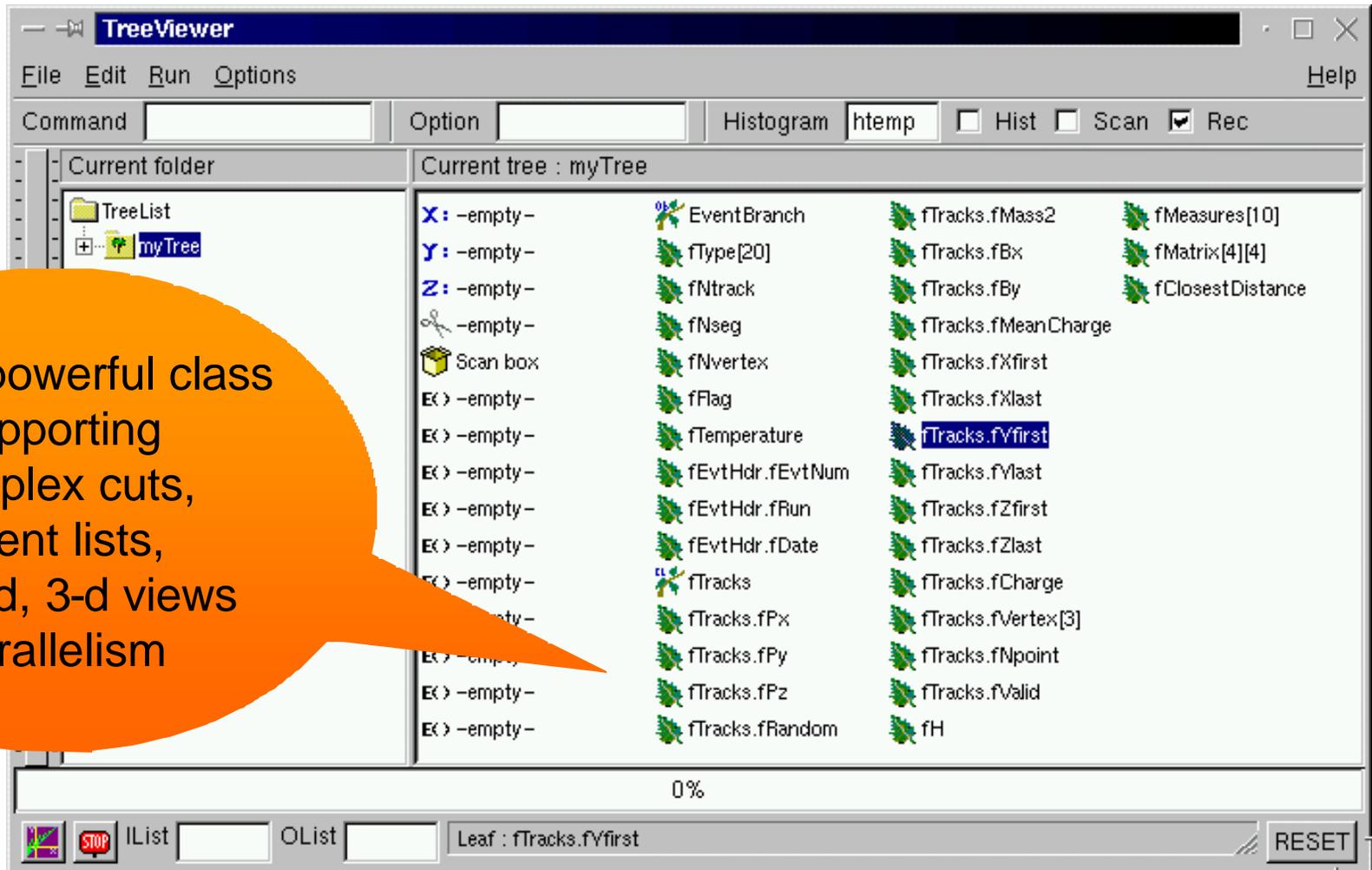
Trees and Folders

- A complete Tree can be generated automatically from an existing Folder structure
- One Branch can be generated from a Folder
- A Folder structure can be automatically rebuilt from a Tree file

The Tree Viewer & Analyzer

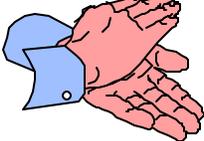


A very powerful class supporting complex cuts, event lists, 1-d, 2-d, 3-d views parallelism





New TTreeFormula

- The class `TTreeFormula` has been extended to process the new `TBranchElement` class. **Thanks Philippe Canal.** 
- Can access data members in an object hierarchy in split and/or non-split modes
- support multiple levels of calls to member functions in split or/and no-split modes.
- Support complex classes in `TClonesArray`
- Examples

New TTreeFormula



```
class Event : public TObject {
private:
    char          fType[20];
    Int_t         fNtrack;
    Int_t         fNseg;
    Int_t         fNvertex;
    UInt_t        fFlag;
    Float_t       fTemperature;
    EventHeader   fEvtHdr;
    TClonesArray *fTracks;           //->
    TH1F          *fH;               //->
    Int_t         fMeasures[10];
    Float_t       fMatrix[4][4];
    Float_t       *fClosestDistance; // [fNvertex]
    static TClonesArray *fgTracks;
    static TH1F      *fgHist;
// ... list of methods
...
    ClassDef(Event,1) //Event structure
};
```

```
Event *event = 0;
```

```
TTree T("T");
```

```
T.Branch("event","Event",&event,32000,2);
```

New TTreeFormula



```
// Data members and methods
1. tree->Draw ("fNtrack");
2. tree->Draw ("event.GetNtrack()");
3. tree->Draw ("GetNtrack()");

4. tree->Draw ("fH.fXaxis.fXmax");
5. tree->Draw ("fH.fXaxis.GetXmax()");
6. tree->Draw ("fH.GetXaxis().fXmax");
7. tree->Draw ("GetHistogram().GetXaxis().GetXmax()");

// expressions in the selection paramter
8. tree->Draw ("fTracks.fPx", "fEvtHdr.fEvtNum%10 == 0");
9. tree->Draw ("fPx", "fEvtHdr.fEvtNum%10 == 0");

// Two dimensional arrays
// fMatrix is defined as:
// Float_t fMatrix[4][4]; in Event class
10. tree->Draw ("fMatrix");
11. tree->Draw ("fMatrix[ ][ ]");
12. tree->Draw ("fMatrix[2][2]");
13. tree->Draw ("fMatrix[ ][0]");
14. tree->Draw ("fMatrix[1][ ]");
```

New TTreeFormula



```
// using two arrays
// Float_t fVertex[3];      in Track class
15.  tree->Draw ("fMatrix - fVertex");
16.  tree->Draw ("fMatrix[2][1] - fVertex[5][1]");
17.  tree->Draw ("fMatrix[ ][1] - fVertex[5][1]");
18.  tree->Draw ("fMatrix[2][ ] - fVertex[5][ ]");
19.  tree->Draw ("fMatrix[ ][2] - fVertex[ ][1]");
20.  tree->Draw ("fMatrix[ ][2] - fVertex[ ][ ]");
21.  tree->Draw ("fMatrix[ ][ ] - fVertex[ ][ ]");

// variable length arrays
22.  tree->Draw ("fClosestDistance");
23.  tree->Draw ("fClosestDistance[fNvertex/2]");

// mathematical expressions
24.  tree->Draw ("sqrt(fPx*fPx + fPy*fPy + fPz*fPz)");

// strings
25.  tree->Draw ("fEvtHdr.fEvtNum", "fType==\"type1\" ");
26.  tree->Draw ("fEvtHdr.fEvtNum", "strstr(fType, \"1\" ");
```

New TTreeFormula



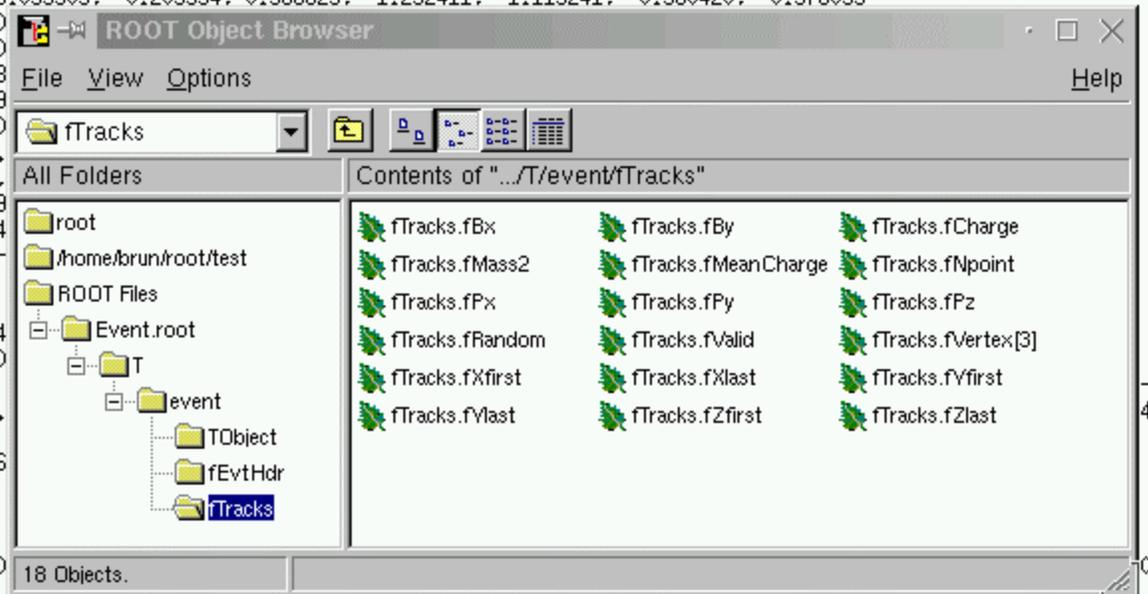
```
// Where fPoints is defined in the track class:
//      Int_t  fNpoint;
//      Int_t *fPoints; [fNpoint]
27.  tree->Draw("fTracks.fPoints");
28.  tree->Draw("fTracks.fPoints
              - fTracks.fPoints[][fAvgPoints]");
29.  tree->Draw("fTracks.fPoints[2][]
              - fTracks.fPoints[][55]");
30.  tree->Draw("fTracks.fPoints[][]
              - fTracks.fVertex[][]");

// Selections
31.  tree->Draw("fValid&0x1",
              "(fNvertex>10) && (fNseg<=6000)");
32.  tree->Draw("fPx", "(fBx>.4) || (fBy<=-.4)");
33.  tree->Draw("fPx",
              "fBx*fBx*(fBx>.4) + fBy*fBy*(fBy<=-.4)");
34.  tree->Draw("fVertex", "fVertex>10")
35.  tree->Draw("fPx[600]")
36.  tree->Draw("fPx[600]", "fNtrack>600")
```

read/query Trees without the classes



```
root [0] TFile f("Event.root")
Warning in <TClass::TClass>: no dictionary for class Event is available
Warning in <TClass::TClass>: no dictionary for class EventHeader is available
Warning in <TClass::TClass>: no dictionary for class Track is available
root [1] T.Show(45)
=====> EVENT:45
fUniqueID      = 0
fBits          = 50331648
fType[20]      = 116 121 112 101 48 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
fNtrack        = 609
fNseg          = 6066
fNvertex       = 7
fFlag          = 1
fTemperature   = 20,529432
fEvtHdr.fEvtNum = 45
fEvtHdr.fRun   = 200
fEvtHdr.fDate  = 960312
fTracks        = 609
fTracks.fPx    = -0,077692, -2,625842, 1,130895, 3,095905, -0,209354, 0,988825, -1,232411, -1,119241, -0,580420, -0,378055
fTracks.fPy    = 0,332117, 0,482258, -0,789722, -0,
fTracks.fPz    = 0,341083, 2,669760, 1,379341, 3,0
fTracks.fRandom = 529,431580, 529,431580, 529,43158
fTracks.fMass2 = 8,900000, 8,900000, 8,900000, 8,9
fTracks.fBx    = -0,042621, 0,069631, -0,141984, 0
fTracks.fBy    = 0,001728, 0,116303, 0,046655, -0,
fTracks.fMeanCharge = 0,001209, 0,001738, 0,006828,
fTracks.fXfirst = 2,092409, 0,549266, -1,027804, -9
fTracks.fXlast  = 5,749741, 15,938519, 0,260784, 14
fTracks.fYfirst = 10,398095, 2,455857, 16,579025, -
fTracks.fYlast  = -7,106707, 13,617641, 12,946399,
fTracks.fZfirst = 56,846382, 39,277451, 47,035370,
fTracks.fZlast  = 211,758606, 213,753479, 217,62054
fTracks.fCharge = 0,000000, 0,000000, 1,000000, 0,0
fTracks.fVertex[3] = -0,181014 0,105711 -20,070364
0,034267 0,096083 -3,769124 , -0,119004 0,038680 -1,
79879 , -0,165064 0,044276 -9,960069
fTracks.fNpoint = 64, 60, 66, 61, 62, 61, 62, 64, 6
fTracks.fValid  = 1, 0, 1, 0, 1, 1, 0, 1, 0, 1
fH              = (TH1F*)8a93ed0
fMeasures[10]   = -2 0 5 2 5 1 14 13 9 11
fMatrix[4][4]   = -0,625079 0,530725 -0,786688 0,00
,000000 0,000000
fClosestDistance = 1,441706 1,708780 -0,655489 1,539576 0,804130 0,048241 -0,594909
root [2] new TBrowser
(class TBrowser*)0x88b7460
root [3]
```



read/query Trees without the classes



```
root [0] TFile f("Event.root")
Warning in <TClass::TClass>: no dictionary for class Event is available
Warning in <TClass::TClass>: no dictionary for class EventHeader is available
Warning in <TClass::TClass>: no dictionary for class Track is available

root [1] T.Draw("fPx")

root [2] TLeaf *leaf = T.GetLeaf("fNtrack")

root [3] T.GetEntry(123)

root [4] leaf->GetValue()
(const Double_t)5.990000000000000000e+02

root [5] leaf->GetBranch()->GetEntry(89)

root [6] leaf->GetValue()
(const Double_t)6.070000000000000000e+02
```



Automatic Code Generators

- Data sets can be analyzed by the same classes used to store the data.
- However, one must be able to read the data without these original classes. The classes may not be available after some time.
- Root provides two utilities to generate a class skeleton to read the data, still preserving the attribute names, types and the structure.
 - `TTree::MakeClass`
 - `TTree::MakeSelector`

This point is important.
You can always analyze
a data set even if you have lost
the class(es) that generated
this data set



TTree::MakeClass

- `tree.MakeClass("myClass");` generates two files: `myClass.h` and `myClass.C`
- `myClass.h` contains the class declaration and member functions code that is selection invariant.
- `myClass.C` contains an example of empty loop where one can insert the analysis code
- Usage:
 - `root > .L myClass.C` or `.L myClass.C++`
 - `root > myClass xx;`
 - `root > xx.Loop();`

Use the interpreter

Use the native compiler
The file `myClass.C`
is automatically compiled
and linked !!



TTree::MakeSelector

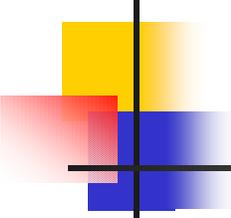
- `tree.MakeSelector("myClass");` generates two files: `myClass.h` and `myClass.C` that can work in a parallel system like PROOF. The event loop is not under user control.
- `myClass.h` contains the class declaration and member functions code that is selection invariant.
- `myClass.C` contains the skeleton of 4 functions: `Begin`, `ProcessCut`, `ProcessFill`, `Terminate`.
- Usage:
 - `root > tree.Process("myClass.C");`
 - `root > chain.Process("myClass.C++");`

Macro is automatically compiled and linked



Folders: in a nutshell

- The class `TFolder` has been in ROOT since quite some time.
- A Folder structure can be used as a white board facility to minimize dependencies between classes via the Folder naming scheme
- User classes/collections in Folders facilitate the documentation and inspection.
- Trees can be generated automatically from Folders.



Some analogy with the past



- At the beginning of computing, communication via Subroutine arguments: No global state
- Labelled Common Blocks
- ZEBRA/BOS solved at least 2 problems
 - Dynamic structures
 - Communication between modules only via the ZEBRA store. Eg, banks from a simulation program could be read in a reconstruction program
- Experience with a large variety of C++ applications indicates a “common-block like” approach.

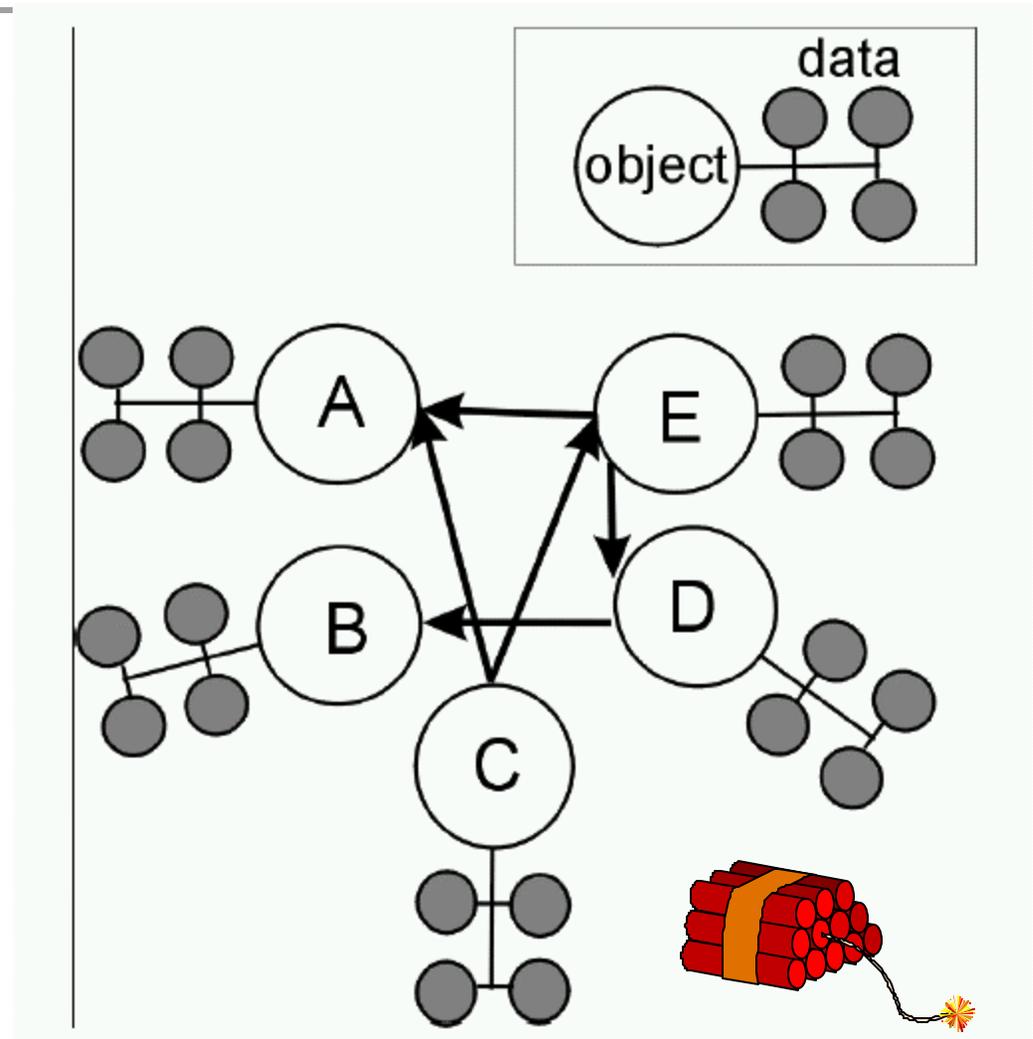


Why Folders ?

This diagram shows a system without folders. The objects have pointers to each other to access each other's data.

Pointers are an efficient way to share data between classes. However, a direct pointer creates a direct coupling between classes.

This design can become a very tangled web of dependencies in a system with a large number of classes.



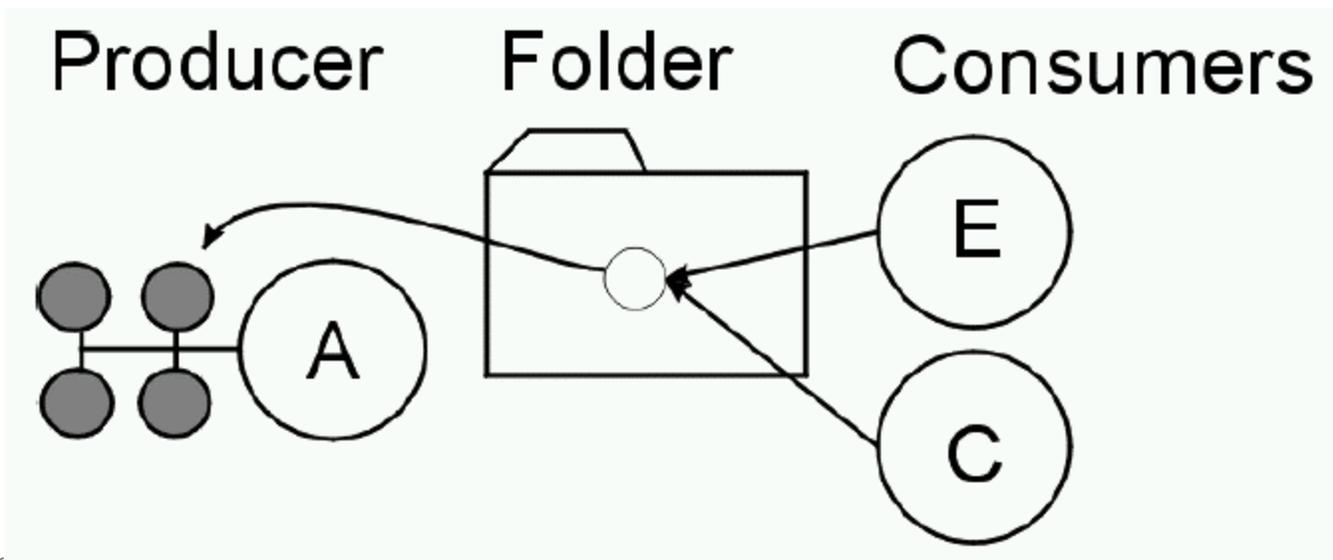


Why Folders ?

In the diagram below, a reference to the data is in the folder and the consumers refer to the folder rather than each other to access the data.

The naming and search service provided by the ROOT folders hierarchy provides an alternative. It loosely couples the classes and greatly enhances I/O operations.

In this way, folders separate the data from the algorithms and greatly improve the modularity of an application by minimizing the class dependencies.



Posting Data to a Folder

(Producer)



- No changes required in user class structure.
- Build a folder structure with:
 - `TFolder::AddFolder(TFolder *)`
- Post objects or collections to a Folder with:
 - `TFolder::Add(TObject*)`
- A TFolder can contain other folders or any TObject descendents. In general, users will not post a single object to a folder, they will store a collection or multiple collections in a folder. For example, to add an array to a folder:
 - `TObjArray *array;`
 - `run_mc->Add(array);`

Reading Data from a Folder

(Consumer)



One can search for a folder or an object in a folder using the **TROOT::FindObjectAny** method. **FindObjectAny** analyzes the string passed as its argument and searches in the hierarchy until it finds an object or folder matching the name. With **FindObjectAny**, you can give the full path name, or the name of the folder. If only the name of the folder is given, it will return the first instance of that name.

```
conf = (TFolder*)gROOT-  
>FindObjectAny("/alroot/Run/Configuration");  
or
```

```
conf = (TFolder*)gROOT->FindObjectAny("Configuration");
```

A string-based search is time consuming. If the retrieved object is used frequently or inside a loop, you should save a pointer to the object as a class data member.

Use the naming service only in the initialization of the consumer class.

Example: Alice folders

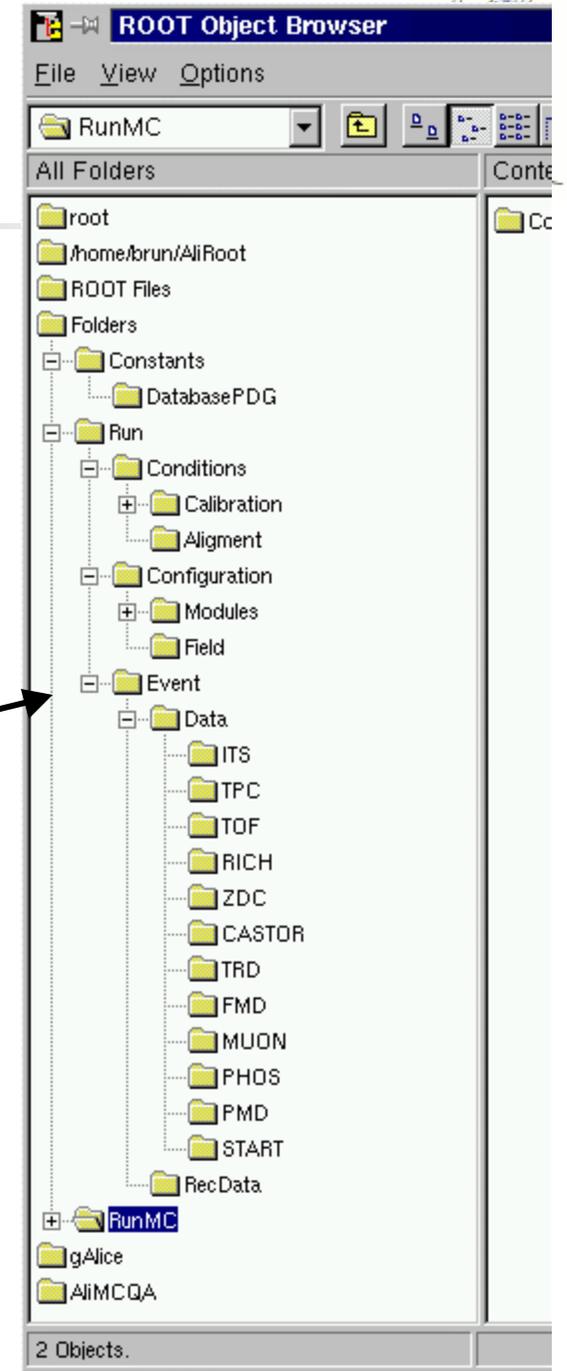
A ROOT Tree can be automatically generated from the folder, eg:

```
TTree T("T","/Event");
```

```
T.Fill();
```

```
T.Show();
```

This statement generates a Tree with 325 branches





Tasks

In the same way that Folders can be used to organize the data, one can use the class **TTask** to organize a hierarchy of algorithms.

Tasks can be organized into a hierarchical tree of tasks and displayed in the browser. The **TTask** class is the base class from which the subtasks are derived. To give a task functionality, you need to subclass the **TTask** class and override the **Exec** method.

Each **TTask** derived class may contain other **TTasks** that can be executed recursively, such that a complex program can be dynamically built and executed by invoking the services of the top level task or one of its subtasks.

```
TTask *run    = new MyRun("run","Process one run");  
TTask *event = new MyEvent("event","Process one event");
```

Use the **TTask::Add** method to add a subtask to an existing **TTask**.

To execute a **TTask**, you call the **ExecuteTask** method.

ExecuteTask will recursively call:

- TTask::Exec** method of the derived class
- TTask::ExecuteTasks** to execute for each task the list of its subtasks.



Execute and Debug Tasks

If the top level task is added to the list of ROOT browse-able objects, the tree of tasks can be visualized by the ROOT browser. To add it to the browser, get the list of browse-able objects first and add it to the collection.

```
gROOT->GetListOfBrowsables()->Add(alroot,"alroot");
```

The browser can be used to start a task, set break points at the beginning of a task or when the task has completed. At a breakpoint, data structures generated by the execution up this point may be inspected asynchronously and then the execution can be resumed by selecting the "Continue" function of a task.

A Task may be active or inactive (controlled by `TTask::SetActive`). When a task is inactive, its sub tasks are not executed.

A Task tree may be made persistent, saving the status of all the tasks.

```

// Example of TTasks.
// Create a hierarchy of objects derived from TTask in library Mytasks
// Show the tasks in a browser.
// To execute a Task, use the context context menu and select
// the item "ExecuteTask"
// see also other functions in the TTask context menu, such as
// -setting a breakpoint in one or more tasks
// -enabling/disabling one task, etc

```

```

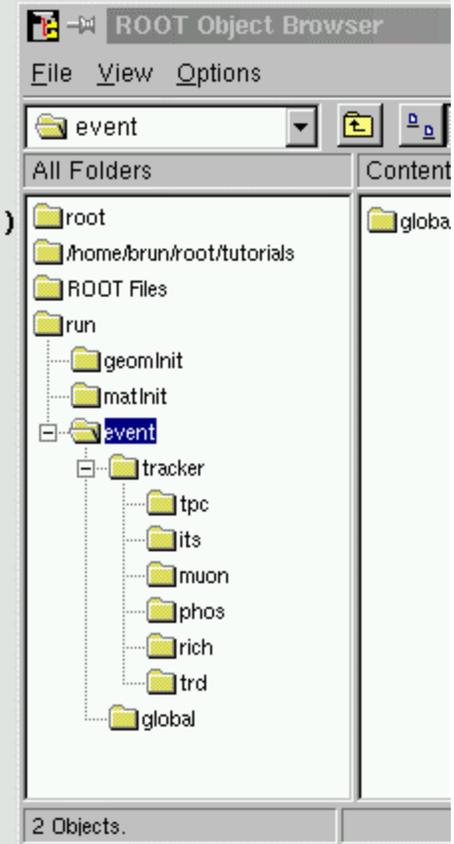
void tasks()
{
  gROOT->ProcessLine(".L MyTasks.cxx");

  TTask *run      = new MyRun("run", "Process one run");
  TTask *event    = new MyEvent("event", "Process one event");
  TTask *geomInit = new MyGeomInit("geomInit", "Geometry Initialisation");
  TTask *matInit  = new MyMaterialInit("matInit", "Materials Initialisation");
  TTask *tracker  = new MyTracker("tracker", "Tracker manager");
  TTask *tpc      = new MyRecTPC("tpc", "TPC Reconstruction");
  TTask *its      = new MyRecITS("its", "ITS Reconstruction");
  TTask *muon     = new MyRecMUON("muon", "MUON Reconstruction");
  TTask *phos     = new MyRecPHOS("phos", "PHOS Reconstruction");
  TTask *rich     = new MyRecRICH("rich", "RICH Reconstruction");
  TTask *trd     = new MyRecTRD("trd", "TRD Reconstruction");
  TTask *global   = new MyRecGlobal("global", "Global Reconstruction");

  run->Add(geomInit);
  run->Add(matInit);
  run->Add(event);
  event->Add(tracker);
  event->Add(global);
  tracker->Add(tpc);
  tracker->Add(its);
  tracker->Add(muon);
  tracker->Add(phos);
  tracker->Add(rich);
  tracker->Add(trd);

  gROOT->GetListOfTasks()->Add(run);
  gROOT->GetListOfBrowsables()->Add(run);
  new TBrowser;
}

```



Folders/Tasks Summary



- Folders minimize coupling between classes
- Folders are browsable. Good for documentation and code reviews.
- Trees generated from folders

- Tasks are browsable. Good for documentation and code reviews.
- Tasks help in understanding a program hierarchy
- Tasks encourage common behaviors

