ROOT Tutorials – Session 3

Internals of ROOT

Fons Rademakers

What is ROOT?

- The ROOT system is an Object Oriented framework for large scale data handling applications
  - Written in C++
  - Provides, among others,
    - An efficient hierarchical OO database
    - A C++ interpreter
    - Advanced statistical analysis (multi dimensional histogramming, fitting, minimization and cluster finding algorithms)
    - Visualization tools
    - And much, much more (GUI, geometry, networking, image processing, …)
  - The user interacts with ROOT via a graphical user interface, the command line or batch scripts
  - The command and scripting language is C++, thanks to the embedded CINT C++ interpreter and large scripts can be compiled and dynamically loaded

Prehistory

- In the beginning there was PAW
  - HBOOK
  - ZEBRA
  - KUIP
  - COMIS
  - SIGMA
- Mini/Micro-DST analysis was done using Ntuples
  - Ntuples are basically simple tables
  - Only basic types
  - No data structures
  - No cross reference between Ntuples
  - Successful because simple and efficient
- Dead-end
  - No way to grow to more complex data structures
  - Difficult to extend
  - Expensive to maintain
  - Too many languages: Fortran, KUIP, SIGMA

Main Goals for New System

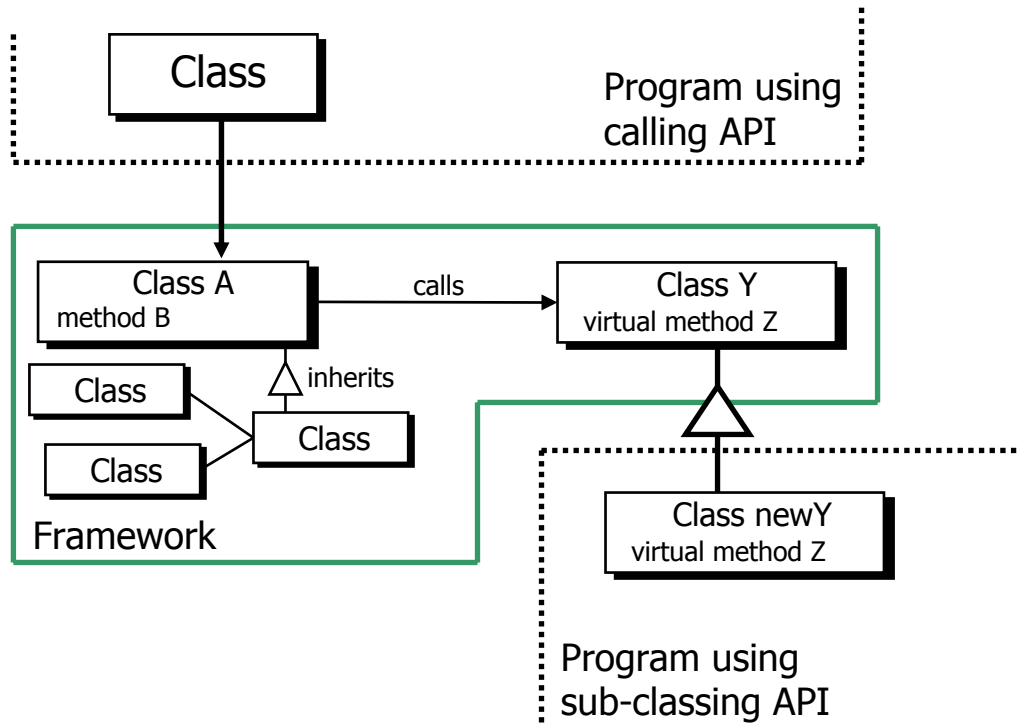
- Being able to support full data analysis chain
  - Raw data, DSTs, mini-DSTs, micro-DSTs
- Being able to handle complex structures
  - Complete objects
  - Object hierarchies
- Support at least the PAW data analysis functionality
  - Histogramming
  - Fitting
  - Visualization
- Only one language
  - C++
- Better maintainable
  - Use OOP
- Make the system extensible
  - Use OO framework technology

Object Oriented Frameworks

Frameworks

- A framework is a collection of cooperating classes that make up a reusable design solution for a given problem domain.
- There are three main differences between frameworks and class libraries:
  - **Behavior versus protocol**. Class libraries are essentially collections of behaviors that you can call when you want those individual behaviors in your program. A framework, on the other hand, provides not only behavior but also the protocol or set of rules that govern the ways in which behaviors can be combined.
  - **Don't call us, we'll call you**. With a class library, the code the programmer writes instantiates objects and calls their member functions. With a framework a programmer writes code that overrides and is called by the framework. The framework manages the flow of control among its objects. This relationship is expressed by the principle: ``Don't call us, we'll call you".
  - **Implementation versus design**. With class libraries programmers reuse only implementations, whereas with frameworks they reuse design. A framework embodies the way a family of related classes work together.

Calling API vs Sub-classing API



Advantages of Frameworks

- The benefits of frameworks can be summarized as follows:
  - **Less code to write**. Much of the program's design and structure, as well as its code, already exist in the framework
  - **More reliable and robust code**. Code inherited from a framework has already been tested and integrated with the rest of the framework
  - **More consistent and modular code**. Code reuse provides consistency and common capabilities between programs, no matter who writes them. Frameworks also make it easier to break programs into smaller pieces
  - **More focus on areas of expertise**. Users can concentrate on their particular problem domain. They don't have to be experts at writing user interfaces, graphics, or networking to use the frameworks that provide those services

ROOT Overview

The Core ROOT Team

Project History

- Jan 95: Thinking/writing/rewriting/???
- November 95: Public seminar, show Root 0.5
- Spring 96: decision to use CINT
- Jan 97: Root version 1.0
- Jan 98: Root version 2.0
- Mar 99: Root version 2.21/08 (1st Root workshop FNAL)
- Feb 00: Root version 2.23/12 (2nd Root workshop CERN)
- Mar 01: Root version 3.00/06
- Jun 01: Root version 3.01/05 (3rd Root workshop FNAL)
- Jan 02: Root version 3.02/07 (LCG project starts: RTAGs)
- Oct 02: Root version 3.03/09 (4th Root workshop CERN)
- May 03: Root version 3.05/05
- Winter 03: Root version 3.10/02
- Spring 04: Root version 4

9 years !!

ROOT Statistics – Supported Platforms

- 3 major type of OS'es
  - Unix, Windows, Mac OS X
- 10 different CPU's
  - IA-32, IA-64, Sparc, Alpha, PA-RISC, PowerPC, MIPS, ARM, Opteron, …
- 11 different compilers
  - Gcc, kcc, ecc, icc, CC, cc, VC++, …
- 41 Makefiles
- ./configure; make


```
(pcrdm) [130] ls Makefile.*
Makefile.aix        Makefile.hpuxacc        Makefile.linuxdeb2     Makefile.linuxrh42   Makefile.sgin32egcs
Makefile.aix5       Makefile.hpuxegcs       Makefile.linuxdeb2ppc  Makefile.linuxrh51   Makefile.solaris
Makefile.aixegcs    Makefile.hpuxia64acc    Makefile.linuxia64ecc  Makefile.linuxsuse6  Makefile.solarisCC5
Makefile.alphacxx6  Makefile.hurddeb        Makefile.linuxia64gcc  Makefile.lynxos      Makefile.solarisgcc
Makefile.alphaegcs  Makefile.in             Makefile.linuxia64sgi  Makefile.macosx      Makefile.solariskcc
Makefile.alphakcc   Makefile.linux          Makefile.linuxicc      Makefile.mklinux     Makefile.win32
Makefile.config     Makefile.linuxalphaegcs Makefile.linuxkcc      Makefile.sgicc       Makefile.win32gdk
Makefile.freebsd    Makefile.linuxarm       Makefile.linuxpgcc     Makefile.sgiegcs
Makefile.freebsd4   Makefile.linuxdeb       Makefile.linuxppcegcs  Makefile.sgikcc
```

ROOT Statistics – Available Binaries

- Intel x86 Linux for Redhat9.0.93(Severn) and gcc 3.3, version 4.00/01 (14.3 MB).
  This version should be compatible with Redhat10. NEW
- Intel x86 Linux for Redhat 9.0 and gcc 3.2.2, version 4.00/01 (15.7 MB). NEW
- Intel x86 Linux for Redhat 7.3 and gcc 3.2, version 4.00/01 (15.9 MB). NEW
- Intel x86 Linux for Redhat 7.3 and gcc 2.96, version 4.00/01 (18.1 MB). NEW
- Intel x86 Linux for Redhat 7.3 and gcc 2.95.2, version 4.00/01 (17.1 MB). NEW
- Intel x86 Linux for Redhat 7.2 and Intel's icc 7.1, version 3.10/01 (24.5 MB). NEW
- Intel x86 Linux for Redhat 6.1 (glibc 2.1) and gcc2.95.2, version 3.10/02 (15.6 MB). NEW
- AMD x86 64 (Opteron) Linux (UnitedLinux) and gcc 3.2.2, version 3.05/05 (11.7 MB). NEW
- Intel Itanium Linux for Redhat 7.2 and gcc 2.96, version 3.05/05 (15.5 MB). NEW
- Intel Itanium Linux for Redhat 7.2 and Intel's ecc 7, version 3.05/05 (31.6 MB). NEW
- HP PA-RISC HP-UX 10.20 with aCC (v1.18), version 4.00/01 (21.3 MB). NEW
- HP Itanium HP-UX 11.20 with aCC, version 3.03/07 (22.1 MB).
- Compaq Alpha OSF1 with cxx 6.2, version 4.00/01 (18.3 MB). NEW
- Compaq Alpha OSF1 with cxx 6.2, version 4.00/01 (18.6 MB). NEW
- Compaq Alpha OSF1 with egcs 1.1.2, version 4.00/01 (21.5 MB). NEW
- Compaq Alpha Linux with egcs 1.1.2, version 3.02/06 (11.0 MB).
- Compaq iPAQ PocketPC Linux with gcc 2.95, version 3.05/03 (11.0 MB). NEW
  For more on Linux on iPAQ see www.handhelds.org.
- IBM AIX 4.5 with xlC version 5, version 4.00/01 (17.5 MB, works only on AIX 4.5). NEW
- Sun SPARC Solaris 5.7 with CC5.2, version 4.00/01 (20.2 MB). NEW
  It cannot be used with Solaris 5.6 or 5.8 even using the same compiler version. You must recompile from the source on these two systems.
- Sun SPARC Solaris 5.8 with CC5.2, version 4.00/01 (19.5 MB). NEW
  It cannot be used with Solaris 5.6 or 5.7 even using the same compiler version. You must recompile from the source on these two systems.
- SGI IRIX 6.5 with CC, version 4.00/01 (compiled with -n32) (18.8 MB). NEW
- SGI IRIX 6.5 with g++ 2.95.2, version 4.00/01 (25.2 MB). NEW
- SGI IRIX 6.5 with KCC, version 4.00/01 (18.1 MB). NEW
- LinuxPPC(Suse7.3) gcc 2.95.3, version 3.03/07 (10.5 MB).
  Thanks to Damir Buskulic (buskulic@lapp.in2p3.fr) for building this version.
- MacOS X 10.2 , 10.3 and gcc 3, version 3.10/01 (25.9 MB) NEW
  A variety of versions is kindly maintained by Remi Monmsen at the sourceforge site. For more details, see Remi's list. For more info, you can contactRemi Mommsen.

- WindowsXP/NT/w2000 with CYGWIN and gcc3.2 version 4.00/01 (18.4 MB).
  For more information see Axel Naumann's web site.
  Do not untar in a directory with a name containing blank characters NEW

29 binary tar balls

ROOT Statistics – Distributions and Number of Users

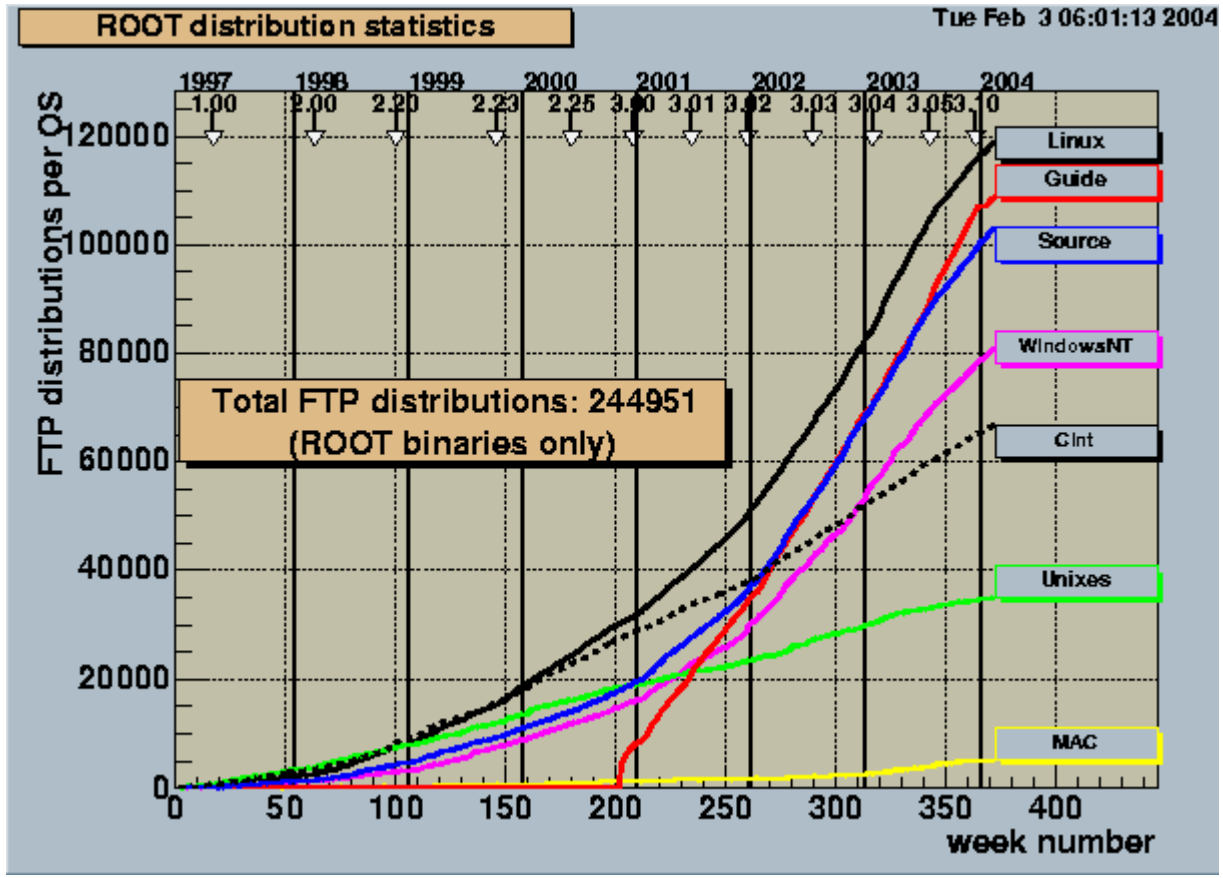
245,000 binaries downloaded

>1,000,000 clicks per month

>100,000 docs in 3 years

3200 registered users

950 users subscribed to roottalk



ROOT Development Process

- We follow an Open Source development model
  - "Release early, release often"
    - Major releases 3-4 times per year
    - Minor releases every 2-3 weeks
    - Daily/nightly builds + regression testing + benchmarking (rootmarks)
  - "Let user feedback drive the development"
    - Bug reporting system
    - Roottalk mailing list and web forum
    - Annual workshop
    - Open cvs repository
    - Let users become developers

ROOT: Framework and Library

- User classes
  - User can define new classes interactively
  - Either using calling API or sub-classing API
  - These classes can inherit from ROOT cla[illegible]es
- Dynamic linking
  - Interpreted code can call compiled code
  - Compiled code can call interpreted code
  - Macros can be dynamically compiled & linked

This is the normal operation mode

Interesting feature for GUIs & event displays

Script Compiler
root > .x file.C++

Dynamic Linking

A Shared Library can be linked dynamically to a running executable module

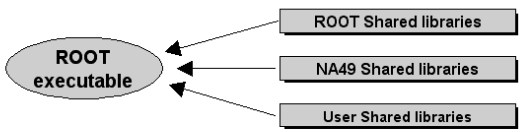
- either via explicit loading,
- or automatically via plug-in manager

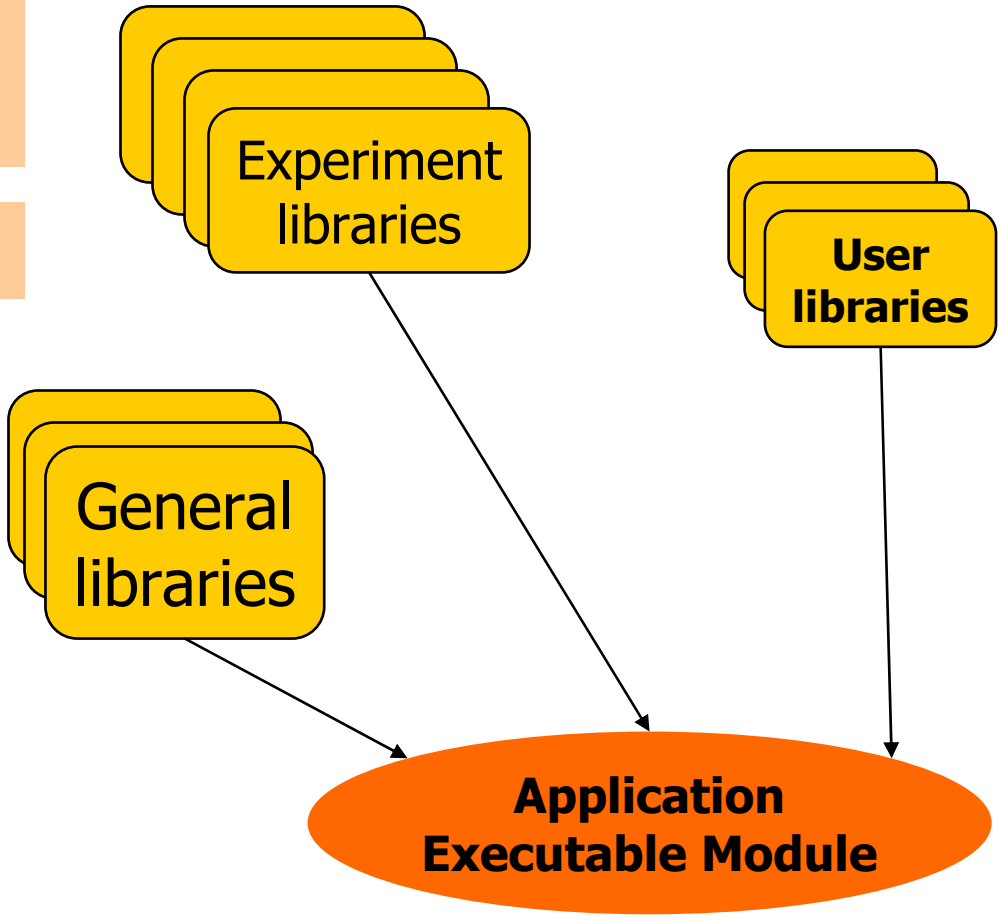
A Shared Library facilitates the development and maintenance phases

Dynamic linking from Shared libraries

The "standard" ROOT executable module can dynamically load user's specific code from shared libraries.

```
Root >  gSystem->Load("libNA49")
Root >  gSystem->Load("libUser")
Root >  T49Event event
Root >  event.xxxxxxx
```

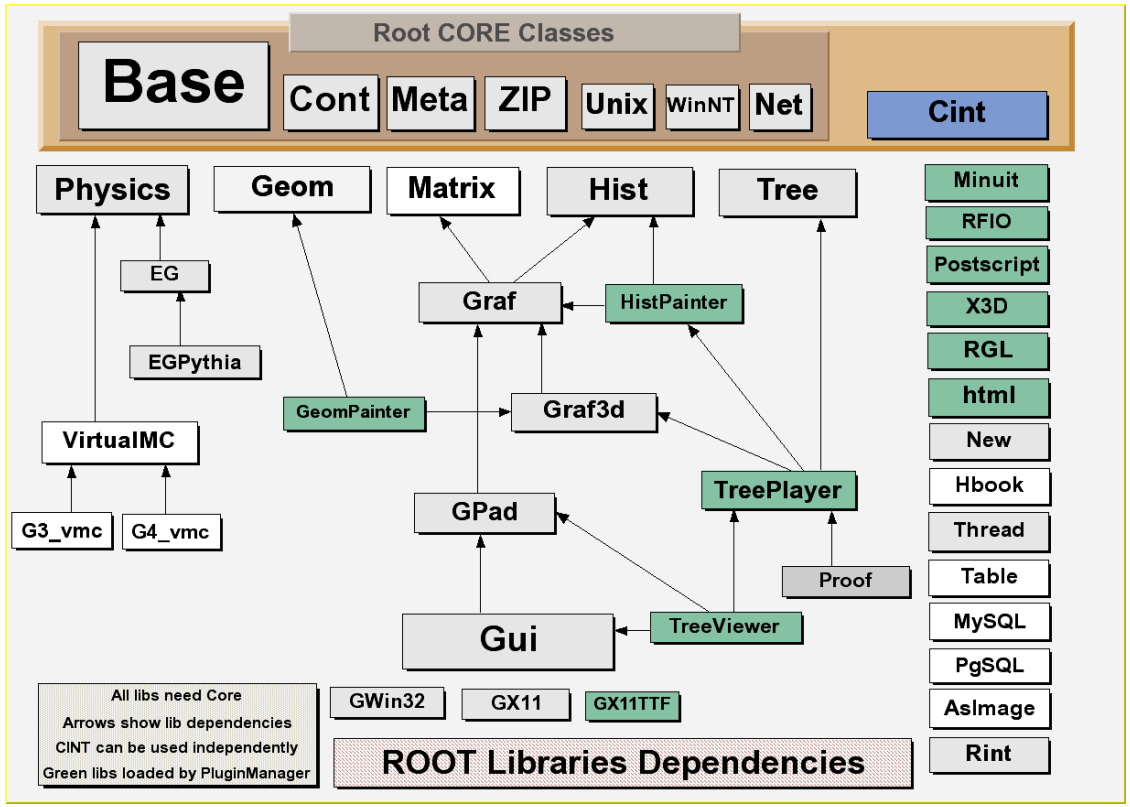




ROOT Library Structure

- ROOT libraries are a layered structure
- The CORE classes are always required (support for RTTI, basic I/O and interpreter)
- The optional libraries (you load only what you use), 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 link time
- Shared libraries reduce the application size
- ROOT shared libraries can be used with other class libraries

The Libraries

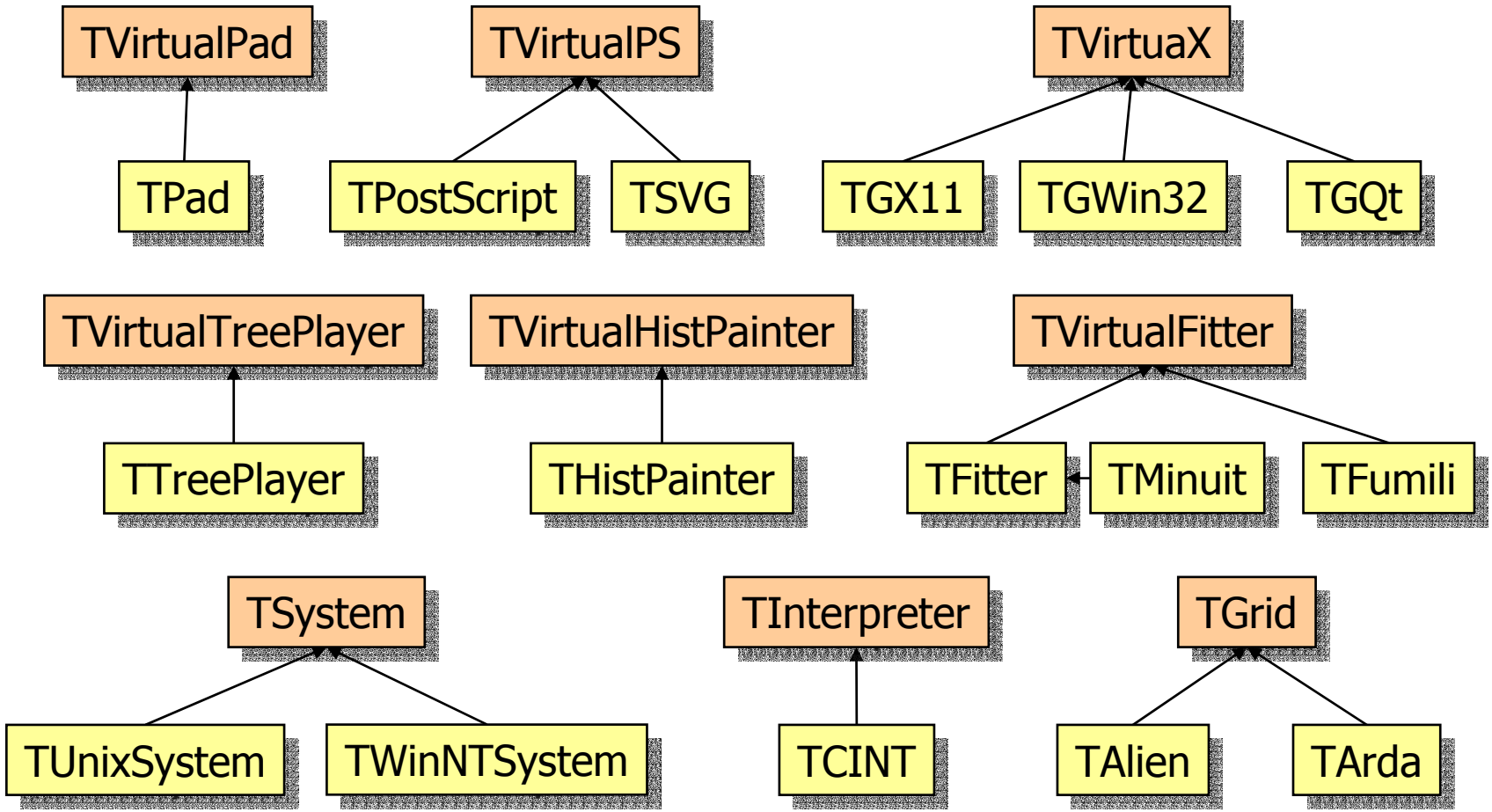


- Over 800 classes
- 1,200,000 lines of code
- CORE (13 Mbytes)
- CINT (3 Mbytes)
- Green libraries linked on demand via plug-in manager (only a subset shown)

ROOT Abstract Interfaces

- The abstract interfaces have two main functions:
  - Define a standard protocol
  - Enhance modularity by minimizing dependencies between classes and shared libraries
- On last count ROOT has 22 abstract interfaces

Example of Abstract Interfaces



CINT Interpreter

CINT

- CINT is a C and C++ interpreter
- Written by Masaharu Goto and available under an Open Source license
- It implements about 95% of ANSI C and 90% of ANSI C++
- It is robust and complete enough to interpret itself (90000 lines of C, 5000 lines of C++)
- Has good debugging facilities
- Has a byte code compiler
- In many cases it is faster than tcl, perl and python

CINT in ROOT

- CINT is used in ROOT:
  - As command line interpreter
  - As script interpreter
  - To generate class dictionaries
  - To generate function/method calling stubs
- The command line, script and programming language become the same
- Large scripts can be compiled for optimal performance

CINT as Interpreter

- CINT is used as command line interpreter:

```
root[0] for (int i = 0; i < 10; i++) printf("Hello\n")
root[1] TF1 *f = new TF1("f", "sin(x)/x", 0, 10)
root[2] f->Draw()
```

- And as script interpreter:

```
bash$ vi script.C
{
   for (int i = 0; i < 10; i++) printf("Hello\n");
   TF1 *f = new TF1("f", "sin(x)/x", 0, 10);
   f->Draw();
}
root[0] .x script.C
```

The Command Line

- The CINT/ROOT command line support emacs style editing
  - ctrl-a, ctrl-e, ctrl-d, left/right arrow keys, etc.
  - Important feature: <TAB> expansion to expand class names, method names, file names, etc.
- Command history is retained between sessions in the file ~/.root_hist
  - Navigation: ctrl-p, ctrl-n, up/down arrow keys
- Everything you type at the prompt is C++
- Except for interpreter escape commands, like
  - .x, .L, .q, .?, etc.
  - Do .? to see all interpreter commands

CINT Debugger

- CINT supports script debugging and tracing:
  - It uses a gdb like command set:
    - .b, .p, .s, .c, .t, etc
  - To trace a script use the .T command
- The byte code compiler can be turned on/off:
  - Off: .O0
  - On: .O1 to .O4
- Again, check available commands with .?

ROOT Infrastructure & Basic Services

The TObject Base Class

- The TObject class provides default behavior and protocol for almost all objects in the ROOT system
- It provides protocol for:
  - Persistency (object I/O)
  - Error handling
  - Inspection
  - Drawing, printing
  - Sorting, hashing

The TROOT Class

- The TROOT object is the main entry point to the system
- It is created as soon as the Core library is being loaded
- It initializes the ROOT system
- It is a singleton, accessible via the global pointer **gROOT**
- Via **gROOT** you can find basically every object created by the system

- Provides many global services

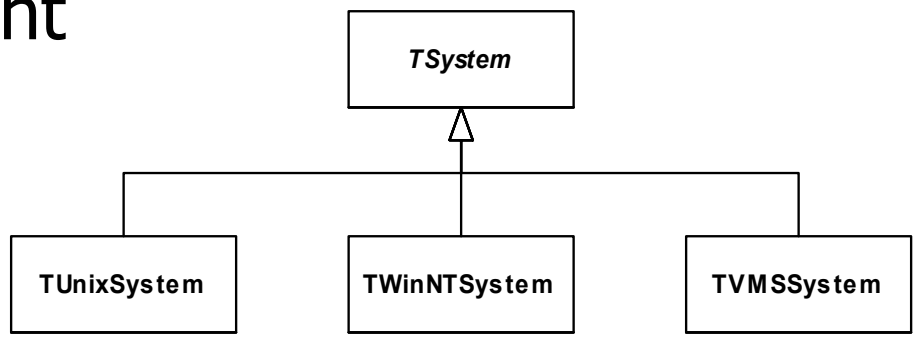
```
TH1F *hpx = (TH1F*) gROOT->FindObject("hpx")
```

ROOT Run Configuration File

- When TROOT is created it also reads the rootrc files:
  - **`$ROOTSYS/etc/system.rootrc`**
  - **`~/.rootrc`**
  - **`./.rootrc`**
- The local one overrides the less local one
- It has the format of a typical "resource" file with a simple syntax
- Have a look at **`$ROOTSYS/etc/system.rootrc`** to see what resources are supported

Operating System Interface

- The underlying OS is abstracted via the TSystem abstract base class
- Accessible via the gSystem singleton
- It allows all ROOT and user code to be OS independent



TSystem Services

- TSystem provides:
  - System event handling
    - event processing and dispatching
    - signal handling (TSignalHandler)
    - file and socket handling (TFileHandler)
    - timer handling (sync, async) (TTimer)
  - Process control
    - fork, exec, wait, …
  - File system access
    - file creation and manipulation
    - directory creation, reading, manipulation

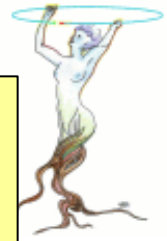
More TSystem Services

- Environment variable manipulation
  - getenv, putenv, unsetenv
- System logging
  - syslog interface
- Dynamic loading
  - load, unload, find symbol, …
- RPC primivitves
  - open, close, option setting, read, write, …
- Please check TSystem carefully for the right methods. Keep your code portable.

ROOT Collections Classes

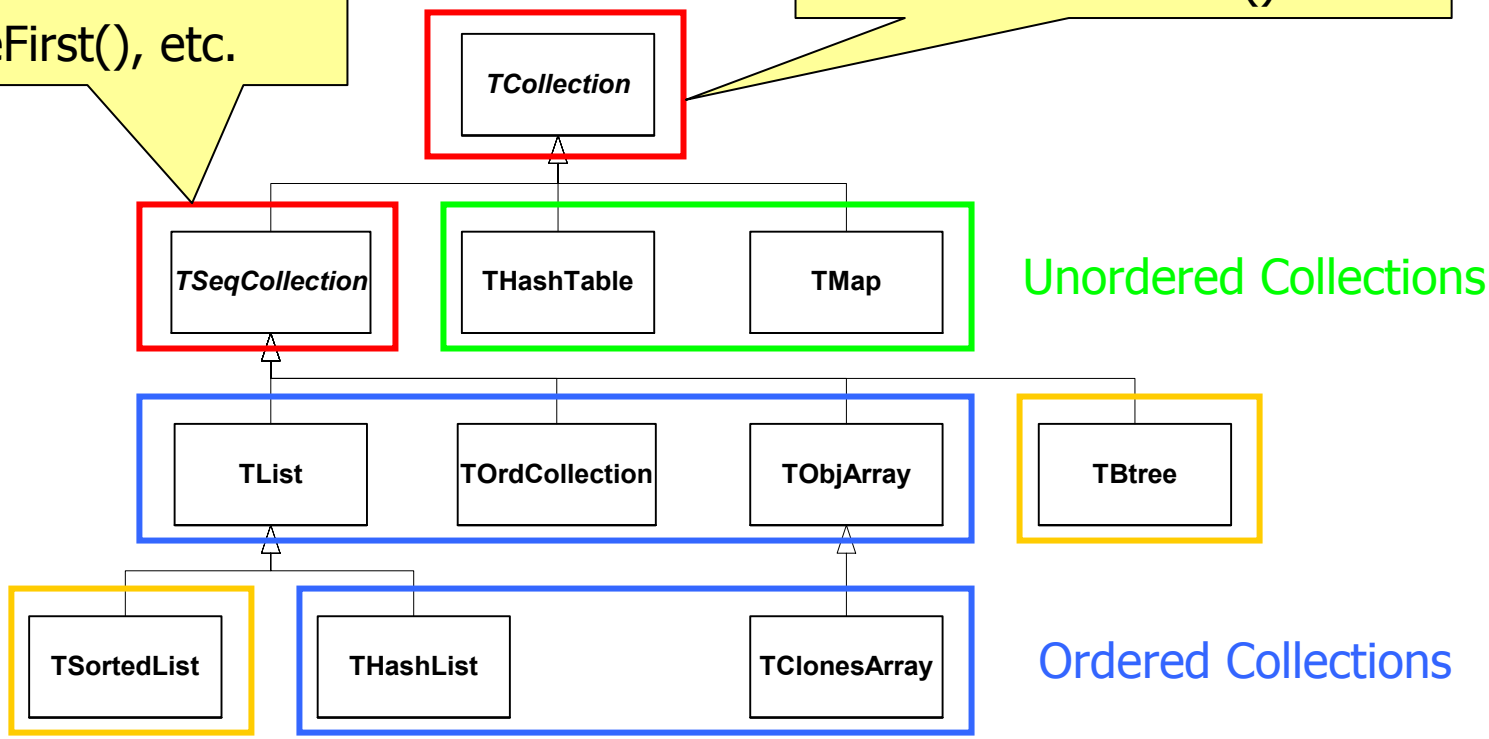
- The ROOT collections are so called *polymorphic collections*
- The collections can contain different types of elements (polymorphism):
  - Elements must be instances of classes
  - Elements must be instances of classes deriving from **TObject**
- Collections themselves derive from TObject. So you can have collections of collections and collections can be made persistent

n Classes



Abstract Base Class
Defines methods like:
AddFirst(), AddLast(),
AddBefore(), AddAfter(),
RemoveFirst(), etc.

Abstract Base Class
Defines methods like:
Add(), Remove(), Clear(),
Delete(), FindObject(),
MakeIterator()



Unordered Collections

Ordered Collections

Sorted Collections

Iterators

- An iterator is used to traverse (walk through) a collection
- Having the iterator separate from the collection allows you to have several iterators on a single collection at the same time
- Each collection has its own associated iterator class:
  - TList TListIter
  - TMap TMapIter
- In general you will use the generic **TIter** wrapper class

TIter: The Generic Iterator

- A TIter object can be used to iterate over any collection

```
TIter it(GetListOfTracks());
while (Track *tr = (Track*) it.Next())
   tr->Fit();
```

```
TIter next(GetListOfTracks());
while (Track *tr = (Track*) next())
   tr->Fit();
```

The magic is in: `TIter::operator()`

```
GetListOfTracks()->ForEach(Track,Fit)();
```

TObject Protocol for Collections

- TObject defines basic protocol for collection elements:
  - IsEqual() used by FindObject(), by default compares addresses
  - IsSortable() used for sorting, by default false
  - Compare() used for sorting, by default not usable
  - Hash() used for hashing, by default address of object
- The collections will call these TObject methods to find, sort and hash elements
- By overriding these methods a class can customize its behavior in a collection

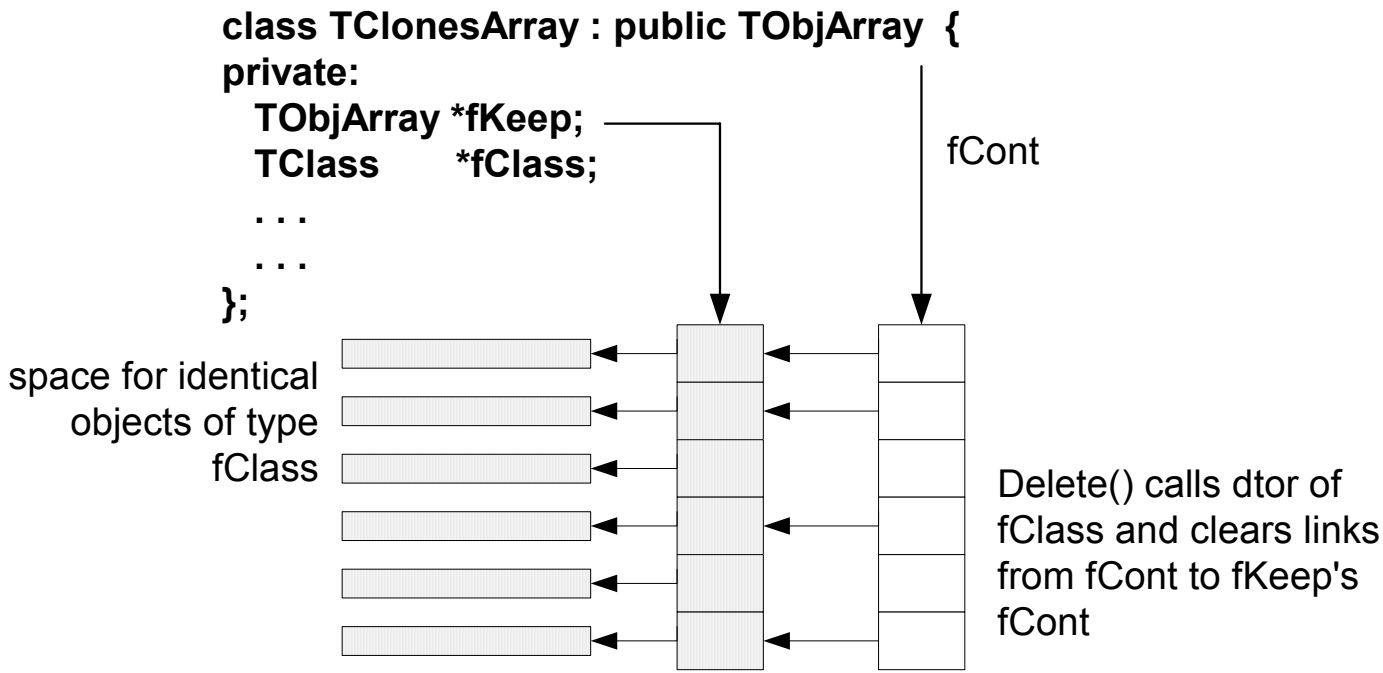
Object Ownership

- The collection classes always store pointers to objects, *never copies of objects*
- It is the user's responsibility to keep track of ownership

```
class Event : public TObject {
   TList *fTracks;  // list of all tracks
   TList *fVertex1; // subset of tracks
   TList *fVertex2; // subset of tracks
   . . .
};
Event::Event()
{
   fTracks = new TList;
   fTracks->SetOwner();
   fVertex1 = new TList;
   fVertex2 = new TList;
}
```

TClonesArray – Array of Identical Objects

A collection specially designed for repetitive data analysis tasks, where normally in a loop many times the same objects are created and deleted



The memory for the objects stored in the array is allocated only once in the lifetime of the clones array

TClonesArray Theory

Considering that a new/delete costs about 70μs, saving O(10^9) new/deletes will save about 19 hours

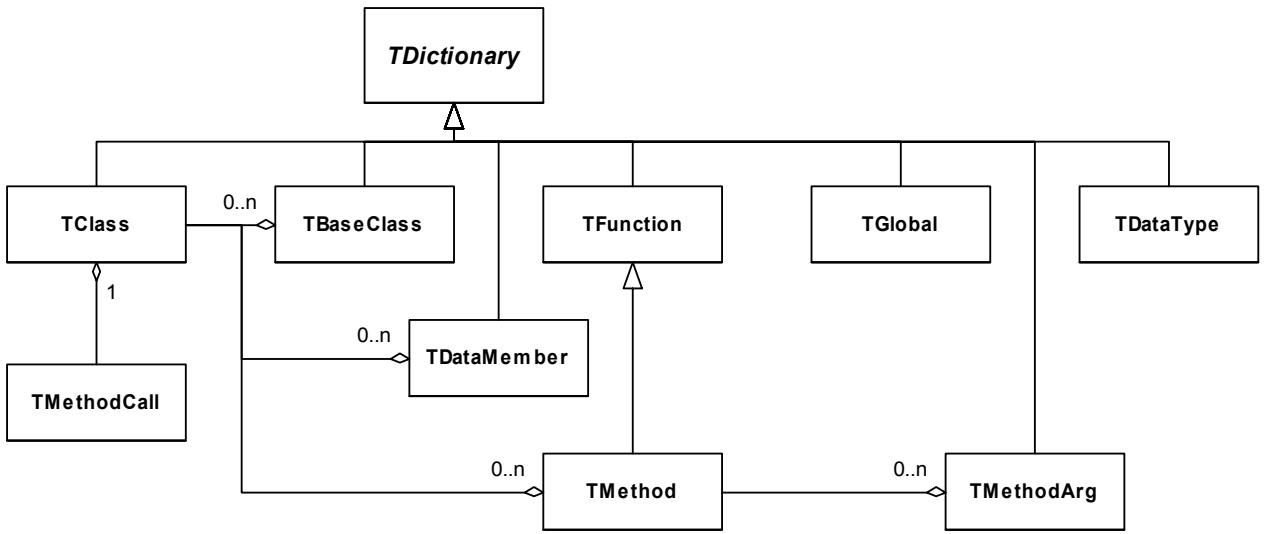
```
TClonesArray a("Track", 10000);
while (Event *ev = (Event *) next()) {          // O(100000)
   for (int i = 0; i < ev->Ntracks(); i++) { // O(10000)
      new(a[i]) Track(x,y,z,. . .);
      . . .
   }
   . . .
   a.Delete();
}
```

Templated Containers and STL

- Templated containers and STL provide type safety at compile time, but
  - They do not solve the problem when the container has to hold a heterogeneous set of objects
- However, ROOT and CINT have no problem with STL containers in user's code
- In ROOT version 4 they are fully supported as first class citizens

ROOT Reflection Classes

- Using the following meta or reflection classes you can find out everything about an object at run-time:



Using Reflection Classes

```
root [0] TLine l
root [1] TClass *c = l.IsA()
root [2] TList *ml = c->GetListOfMethods()
root [3] TIter next(ml)
root [4] TMethod *m
root [5] while (m = (TMethod*)next()) printf("%s\n", m->GetName())
root [6] ml = c->GetListOfDataMembers()
root [7] . . .
```

ROOT Beans

```
{
   TClass *cl = gROOT->GetClass("TLine");
   void *line = cl->New();
   TMethod *m;
   m = (TMethod*) cl->GetListOfMethods()->FindObject("SetX2");
   if (m) {
      // use m->GetListOfMethodArgs() to check argument types
      TMethodCall mc(cl, "SetX2", "10.0");
      mc.Execute(line);
   }
   m = (TMethod*)cl->GetListOfMethods()->FindObject("SetY2");
   if (m) {
      TMethodCall mc(cl, "SetY2", "20.0");
      mc.Execute(line);
   }
   TMethodCall mc(cl, "Draw", "");
   mc.Execute(line);
}
```

ComponentWare

- Using the reflection classes and dynamic library loading it is very simple to build the equivalent of "Java Bean" components
  - Total decoupling, extreme modularity
  - Embedding
  - Flexible I/O
- Only need to agree on a set of strings that components must understand

Class and Object Tables

- At run time one can see all classes for which RTTI is available:

```
gClassTable->Print();
```

- In addition one can see all TObject derived objects that have been created:

```
gObjectTable->Print();
```

- This last feature requires the rootrc option:
  - Root.ObjectStat: 1

Plug-in Manager

- Where are plug-ins used?

```
TFile *rf = TFile::Open("rfio://castor.cern.ch/alice/aap.root")

TFile *df = TFile::Open("dcache://main.desy.de/h1/run2001.root")
```

- For example, to extend the base class TFile to be able to read RFIO files one needs to load the plug-in library libRFIO.so which defines the TRFIOFile class
- Protocol part of the file name URI triggers loading of plug-in. In these cases TRFIOFile and TDCacheFile objects are used, which both derive from TFile

Plug-in Manager

- Previously dependent on “magic strings” in source, e.g. in TFile.cxx:

```
if (!strncmp(name, "rfio:", 5)) {
   if (gROOT->LoadClass("TRFIOFile", "RFIO")) return 0;
   f = (TFile*) gROOT->ProcessLine(Form("new
                  TRFIOFile(\"%s\",\"%s\",\"%s\",%d)",
                  name, option, ftitle, compress));
} else if (!strncmp(name, "dcache:", 6)) {
```

- Adding case or changing strings requires code change and recompilation
- Not user customizable

Plug-in Manager (cont.)

- Plug-in manager solves these problems:

```
TPluginHandler *h;
if ((h = gROOT->GetPluginManager()->FindHandler("TFile", name))) {
   if (h->LoadPlugin() == -1) return 0;
   f = (TFile*) h->ExecPlugin(4, name, option, ftitle, compress);
}
```

- Single if-statement to handle all cases
- No magic strings in code anymore

Plug-in Manager (cont.)

- Magic strings moved to **system.rootrc** file

```
# base class    regexp     plugin class   plugin lib   ctor or factory
Plugin.TFile:   ^rfio:     TRFIOFile      RFIO         "TRFIOFile(const
                                        char*,Option_t*,const char*,Int_t)"
+Plugin.TFile: ^dcache:    TDCacheFile    DCache       "TDCacheFile(const
                                        char*,Option_t*,const char*,Int_t)"
```

- Adding plug-ins or changing strings does not require code change and recompilation
- Can be customized by user in private **.rootrc** file

Plug-in Manager (cont.)

- Currently 34 plug-ins are defined for 21 different (abstract) base classes
- Plug-in handlers can also be registered at run time, e.g.:

```
gROOT->GetPluginManager()->AddHandler("TSQLServer",
   "^sapdb:","TSapDBServer", "SapDB",
   "TSapDBServer(const char*)");
```

- A list of currently defined handlers can be printed using:

```
gROOT->GetPluginManager()->Print();
```