ROOT

Overview

ROOT, Overview

Authors: René BRUN *(Rene.Brun@cern.ch)*
Nenad BUNCIC *(Nenad.Buncic@cern.ch)*
Valery FINE *(Valeri.Faine@cern.ch)*
Fons RADEMAKERS *(Fons.Rademakers@cern.ch)*

Cover Page: Marjana RAPAIC BUNCIC

Table of Contents

ROOT Mission Statement 4
- Why ROOT? 4
- ROOT: Executive Summary 4

Architectural Overview 6
- The Class Categories 6
- The TObject Class 6
- The Class Dictionary and Object Runtime Support 7

The CINT C/C++ Interpreter 8

The ROOT Dictionary 9

The CINT Dictionary Generator 11
- Dictionary Generation for Interactive Access Without I/O and RTTI 11
- Dictionary Generation for Interactive Access Including I/O and RTTI 12

Adding Your Own Classes to ROOT 16

The CINT Interpreter Interface 17
- CINT as Command Line Interpreter 17
- CINT as Macro Processor 18
- ROOT/CINT Extensions to C++ 21

CINT Commands 22

Emacs Style Command Line Editing 23

ROOT Tutorials 24

The ROOT Object I/O System 25
- Why Not Use an OO Data Base Instead? 25
- The Physical File Structure 25
- The Logical File Structure 26
- Support for Class/Schema Evolution 27
- The Input/Output Functions 28
- Compression or No Compression? 28

The ROOT Trees 30

How to Write Objects to a File 32

How to Read Objects from a File 34

How to Draw objects 35

How to Pick objects 36

How to Use the Graphics Editor 37

How to use the Histogram classes 39
- Creating histograms 39
- Filling histograms 39
- Drawing histograms 40

Writing histograms to a file 40
Reading histograms from a file. 40
Fitting histograms 40
Other histogram facilities 40
How to Fit Histograms or Data Points 41
Fitting 1-Dim histograms 41
Fitting non-equidistant data points 42
Fitting in general using the class TMinuit 43
Access to the Fit parameters and results. 44
Fitting with one or more parameters between some bounds 44
How to Create and Fill a Tree 45
Using the serialization method and creating a key 45
Making a Tree with one single branch 46
Making a Tree with many branches 47
Making branches by hand. 47
How to Read a Tree. 48
The simple way via TTree::Draw 48
The general way 48
How to use Chains. 50
How to Debug and understand a class structure 51
How to use the PostScript interface. 54
How to Generate the HTML documentation for user classes. . . .55
How to Compile and Link with the ROOT libraries 56
How to Convert HBOOK files to ROOT .57
Accessing ROOT Files Remotely via a rootd Server 58
Reading ROOT Files via Apache Web Server 60
The ROOT Collection Classes 61
Understanding Collections 62
Types of Collections 62
Using Collection Classes 65
Automatic Document Generation 68
Save Enormous Amount Of Time 68
Navigate Quickly 68
Visualize The Code. 69
Requirements 69
How To Start?. 69
What Else Should I Know?. 70
Advanced Configuration. 70
C++ Coding Conventions 71
Naming conventions 71
Class definition conventions. 71
Avoid raw C types 71
New features like, templates, exception handling, RTTI and namespaces. 71
Using comments to document the code 72
ROOT Input/Output Benchmarks 73
ROOT compared to ZEBRA and HBOOK 75
Status & Availability. 77
Development Status 77
Porting Status 77
Availability 77
Version 1.00 80
Availability 80
The ROOT Team. 82

Overview

ROOT Mission Statement

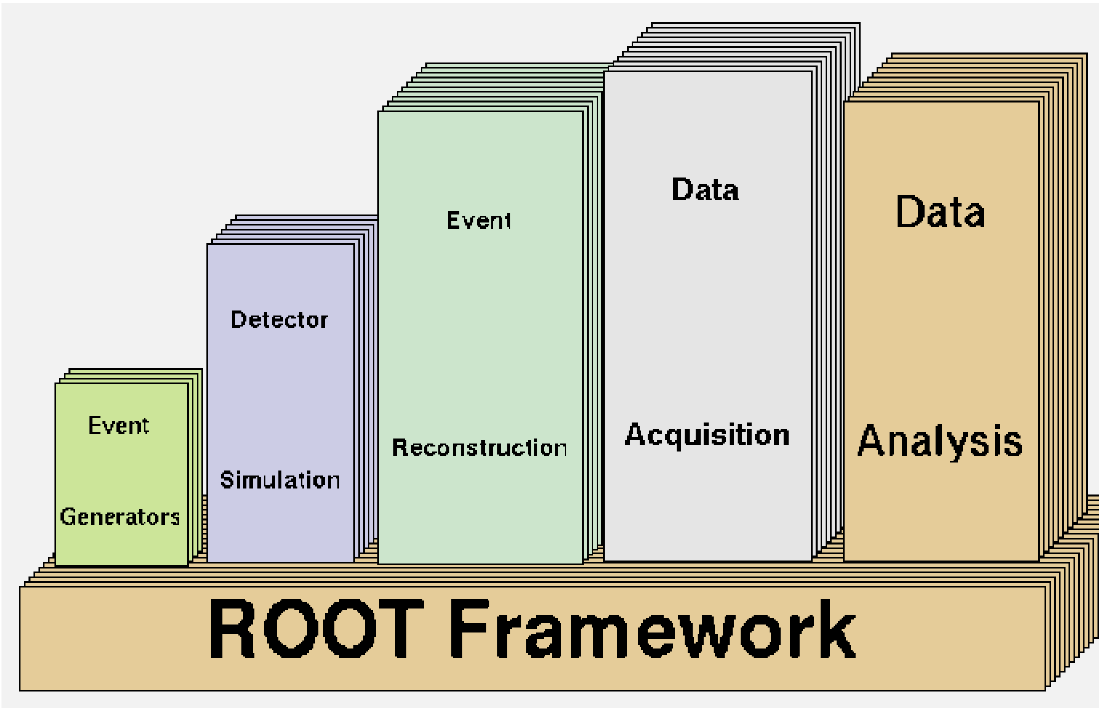
Why ROOT?

Having had many years of experience in developing the interactive data analysis systems PAW and PIAF and the simulation package GEANT, we realized that the growth and maintainability of these products, written in FORTRAN and using some 20 year old libraries, had reached its limits. Although still very popular, these systems do not scale up to the challenges offered by the **LHC**, where the amount of data to be simulated and analyzed is a few orders of magnitude larger than anything seen before.

It became time to re-think our approach to large scale data analysis and simulation and at the same time we had to profit from the progress made in computer science over the past 15 to 20 years. Especially in the area of **Object-Oriented** design and development. Thus was born ROOT.

We started the ROOT project in the context of the NA49 experiment at CERN. NA49 generates an impressive amount of data, about 10 Terabytes of raw data per run. This data rate is of the same order of magnitude as the rates expected to be recorded by the **LHC** experiments. Therefore, NA49 is an ideal environment to develop and test the next generation data analysis tools and to study the problems related to the organization and analysis of such large amounts of data.

With ROOT we try to provide a basic **framework** that offers a common set of features and tools for all domains shown in the figure below.



Currently the emphasis of ROOT is on the **data analysis** domain but thanks to the approach of loosely coupled object-oriented frameworks the system can easily be extended to other domains.

We believe that ROOT is an ideal environment to introduce physicists quickly to the new world of **Objects** and **C++**.

ROOT: Executive Summary

The ROOT system provides a set of OO frameworks with all the functionality needed to handle and analyse large amounts of data in a very efficient way. Having the data defined as a set of objects, specialised storage methods are used to get direct access to the separate attributes of the selected objects, without having to touch the bulk of the data. Included are histograming methods in 1, 2 and 3 dimensions, curve fitting, function evaluation, minimisation, graphics and visualization classes to allow the easy setup of an analysis system that can query and process the data interactively or in batch mode.

Thanks to the builtin CINT C++ interpreter the command language, the scripting, or macro, language and the programming language are all C++. The interpreter allows for fast prototyping of the macros since it removes the time consuming compile/link cycle. It also provides a good environment to learn C++. If more performance is needed the interactively developed macros can be compiled using a C++ compiler.

The system has been designed in such a way that it can query its databases in parallel on MPP machines or on clusters of workstations or high-end PC's. ROOT is an open system that can be dynamically extended by linking external libraries. This makes ROOTa premier platform on which to build data acquisition, simulation and data analysis systems.

Architectural Overview

The backbone of the ROOT architecture is a layered class hierarchy with, currently, around 250 classes grouped in about 21 frameworks divided in 11categories. This hierarchy is organized in a mostly single-rooted class library, that is, most of the classes inherit from a common base class **`TObject`**. While this organization is not very popular in C++, it has proven to be well suited for our needs (and indeed for almost all successful class libraries: Java, Smalltalk, MFC, Be OS, etc.). It enabled the implementation of some essential infrastructure inherited by all descendants of `TObject`. However, we also can have classes not inheriting from `TObject` when appropriate (e.g., classes that are used as built-in types, like `TString`).

The Class Categories

The *basic ROOT classes* contain the most low-level building blocks of ROOT. For example, the `TObject` class, which implements common behaviour for all ROOT classes. The class `TClass` and its helper classes that provide support for extended runtime type information. The storage manager `TStorage` which handles all memory allocation and de-allocation operations and performs basic error checking (memory overwrites, etc.). The class `TFile` which provides a hierachical sequential and direct access persistant object store. The operating system abstraction layer `TSystem` and the concrete OS interfaces `TUnixSystem`, `TWin32System` and `TMac-System` concentrate all OS dependent behaviour, like file system access, dynamic loading and IPC for the three main platforms supported by ROOT.

The *container classes* provide general purpose data structure classes like, arrays, lists, sets, B-trees, maps, etc., which are heavily used in the implementation of ROOT itself.

The *histogram and minimization classes* offer advanced statistical data analysis features, like 1D, 2D and 3D histograming of short, long, float or double values, with fixed or variable bin sizes, profile histograms, data fitting, formula evaluation and minimization.

The *tree and ntuple classes* contain the tree system. The row-wise and column-wise ntuples have been one of the major strengths of the PAW system. Trees extend the concept of ntuples to all complex objects and data structures found on raw data tapes and DSTs. The idea is that the same data model, same language, same style of queries can be used on all data sets in an experiment. Trees are designed to support not only complex objects, but also a very large number of them in a large number of files. Ntuples are simple trees with one branch only.

The *2D graphics classes* contain the low-level graphics primitives, like lines, arrows, rectangles, ellipses, text, etc., but also the higher level constructs like pads and canvases. They also handle basic style and attribute management.

The *3D graphics and detector geometry classes* provide basic 3D graphics primitives, like 3D polylines and 3D polymarkers as well as higher level geometrical shapes (boxes, cones, polygons, tubes, etc.) which can be efficiently assembled into very complex detector geometries.

The *MOTIF graphical user interface classes* contain all the graphical and interactive components found in almost every user interface toolkit, such as buttons, windows, dialogs and menus. Similar classes have also been developed for *Windows/NT and Windows 95*.

The *interactive interface classes and C++ interpreter* allow the construction of interactive applications in which the user has to learn only one language, C++, to communicate with the program. The command language, macro language and programming language are all one and the same.

The *operating system interface* is the sole interface to the operating system. All OS services are handled via the `TSystem` (`TUnixSystem`, `TWinNTSystem` and `TMacSystem`) class.

The *networking classes* provide a simple, but powerful way to construct client/server application. All low level IPC services are provided via the TSystem class.

The *documentation classes* allow the creation of hyperized (in HTML format) C++ header and source files, inheritance trees, class indices, macro's and session transcripts. Thanks to this facility almost everything in the ROOT system can be automatically documented and cross-referenced.

The TObject Class

Most ROOT classes are derived from `TObject`. `TObject` defines protocols (abstract methods) for comparing objects, for object I/O, for graphics hit detection and for notification between objects, to name the most important ones.

The ROOT object I/O facility supports the transfer of arbitrarily complex polymorphic data structures from memory to a ROOT file and vice versa. This functionality is based on the abstract method `Streamer`, which is overridden in subclasses to stream an object's instance variables. Circular structures are linearized, and multiple references to the same object are restored properly. Storing pointers is implemented by an object table, which assigns a unique identifier to each transmitted object. This identifier can be transferred to other address spaces or to permanent storage.

Object I/O needs some information about the type of an object at runtime, because not only the state of an object but also its corresponding class type has to be transmitted. ROOT runtime support could provide enough information about an object's instance variables to implement the `Streamer` method generically in class `TObject`. However, we preferred the approach of a programmer selectively deciding which data members should be written to disk. Data members caching some state of an object that can easily be reconstructed in the `Streamer` method do not have to be transferred to disk. Another example is when variables can be compacted into short words or even single bits (booleans). To aid the programmer we provide a tool to generate automatically a default `Streamer` method.

The case of encountering an unknown class while reading back an object structure leads to the discussion of *dynamic loading and linking*. To handle this case gracefully, ROOT includes a mechanism to load a new class and link it to a running application. This dynamic linking support can be further used to extend a running system. In the case of the NA49 data analysis the library with the experiment specific classes is dynamically linked to the running interactive ROOT program.

The object I/O facility is also used as the standard format for transferring arbitrary data structures to other ROOT based applications running in other address spaces or on other machines. The transparent integration of dynamic linking into the object I/O mechanism allows the copying of instances of classes that are not known in the running application. This feature allows us to develop fairly easily advanced, web like, browsers that could operate on imported ROOT objects (for example, we could refit imported histograms without having to leave the browser).

The Class Dictionary and Object Runtime Support

Even with the upcoming runtime type identification (RTTI) extension for C++, the runtime system does not provide any information about the class structure, the instance variables or the member functions of an object. Consequently, an additional mechanism had to be introduced to gather this information, in order to support `InheritsFrom`, `Inspect` and `Dump` methods, the object I/O facility and the automatic documentation system. ROOT uses the approach of associating with each class (via a static pointer) a special object describing its structure. These descriptors are instances of the class `TClass` which is itself a subclass of `TObject`. `TClass` objects store the following information about a class:

- The name and title of a class.
- The size of an instance in bytes.
- Its parent class(es).
- The names, types and descriptions of its instance variables.
- The names and signatures of its member functions.
- A source code reference to the definition and implementation part of the class.
- The address of the class' object factory method used to create a new object.

Because the C++ runtime system gives no access to type and structure information, the ROOT system uses a dictionary generator called **CINT**. **CINT** parses the class header files and generates a dictionary (in the form of a C++ function). To link the CINT generated dictionary function to a class the programmer only has to add two preprocessor macros to his code. One macro, `ClassDef`, must be placed in the class definition file and the other macro, `ClassImp`, in the implementation file.

Besides as a dictionary generator, CINT is being used in the ROOT system as a command line interpreter and macro processor. Thanks to CINT the ROOT system can offer the user a single language (C++) interface.

The CINT C/C++ Interpreter

CINT is a C/C++ interpreter which is aimed at processing C/C++ scripts. Scripts are programs which perform specific tasks. Generally execution time is not critical, but rapid development is. Using an interpreter the compile and link cycle is dramatically reduced facilitating rapid development. CINT covers about 95% of ANSI C and 85% of C++. CINT, written in ANSI C (about 80000 loc), is solid enough to interpret itself and let the interpreted version execute a program.

CINT makes C/C++ programming enjoyable even for part-time programmers.

CINT is developed by Masaharu Goto who is a hardware/software design consultant in the MPN consulting group of HP Japan. Masaharu is working closely with the ROOT team to integrate CINT seamlessly into ROOT and to further optimize CINT/ROOT.

The ROOT system embeds CINT to be able to execute C++ scripts and C++ command line input. CINT also provides ROOT with extensive RTTI capabilities.

Currently CINT 5.13 is available via anonymous ftp for the following platforms:

Commercial use of CINT requires registeration to Hewlett-Packard Japan. Send e-mail to the author (gotom@jpn.hp.com) describing your name, company or organization, address and purpose of using CINT.

See the following chapters on how CINT is used in ROOT:

Adding Your own Classes to ROOT

CINT as Dictionary Generator

CINT as Command Line and Macro Interpreter

CINT Commands

The ROOT Dictionary

When a ROOT based application is running, the following information is stored in its **memory resident dictionary**:

• a list of all **global functions**

• a list of all **global variables**

• a list of all **classes**

• for each class, the full description of its **data members**:

• data member name

• data member type

• associated comment field

• access information (private, protected or public)

• for each class, the full description of its **member functions**:

• function name

• function return type

• list of parameters and their type

• associated comment field

• access information (private, protected or public)

• the class' title and version as specified in the `ClassDef` macro.

• the class' object creation function `New()`

This dictionary is much more extensive than the RTTI facility as proposed in the (draft) C++ standard document. In the ROOT system the dictionary information is used to support, among others, the following features:

• automatic generation of the `Streamer()` functions used by the `TBuffer` class

• input/output via the `TObject::Write()` and `TKey::Read()` methods

• input/output via the `TTree` and `TNtuple` classes. The automatic generation of containers for each object attribute relies on the data member information in the dictionary.

• the documentation generation class `THtml`

• the context menus. When an object drawn in a pad is picked, a context menu is shown with a list of items corresponding to the member functions of its class that have been declared candidates for a menu. If an item of the context menu is selected, a dialog box appears with the list of parameters of the corresponding member function. When the OK button of the dailog box is clicked, the member function is executed.

• the **object inspector** utility. The `TObject::Inspect()` function generates a graphical dump of an object including clickable hyperlinks for pointers.

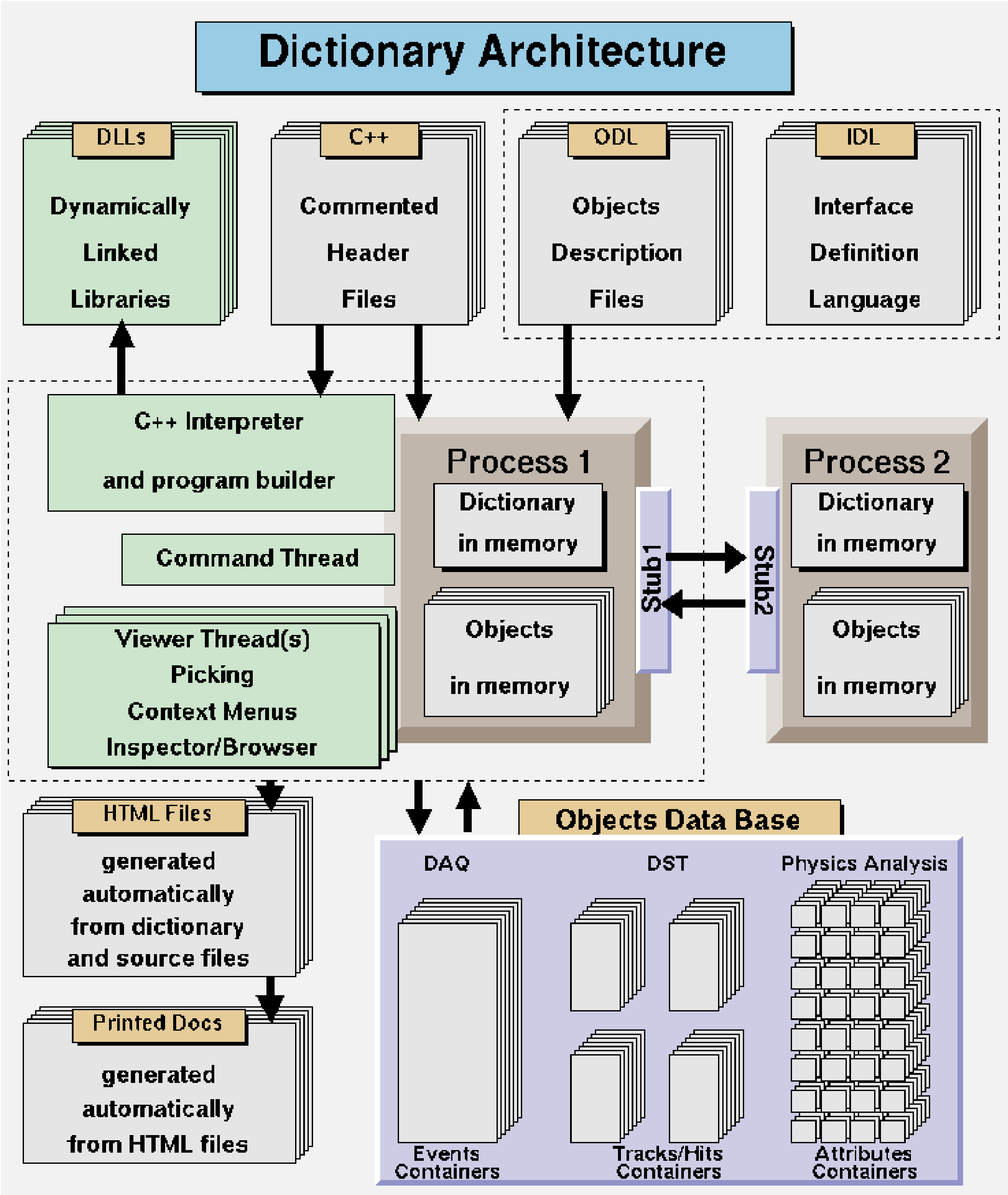
• the browser utility. The ROOT browser can automatically browse through list of objects and list of lists, etc.

• the CINT **C++ interpreter** itself is fundamentaly based on the information stored in the dictionary

• the ROOT **dynamic linker** uses the dictionary information to dynamically link classes in shared libraries or DLLs. This gives the possibility to interactively build a program from existing components in these libraries. With this approach, the link time is considerably reduced and the executable modules are much smaller.

• **Inter-Process Communication** (IPC) is based on the object serializer facility provided by the `Streamer()` method

• The picture below gives a schematic overview of the pervasive usage of the dictionary in ROOT based systems:



The dictionary is generated by the CINT Dictionary Generator using the C++ header files (see picture).

ROOT does not require files generated from the Object Description Language **ODL** as is the case for some Object Oriented Data Bases.

ROOT does not require files generated from the Interface Definition Language **IDL** as is the case for some inter-process communications tools like **CORBA**.

Note that the ROOT classes and the ROOT dictionary functions have been generated in view of multi-threaded applications. In the Windows/NT and Windows95 environment, ROOT already runs with several threads of execution.

The CINT Dictionary Generator

To integrate your classes in the ROOT system and to benefit from the advanced RTTI and I/O features of ROOT and to get access to your classes via the C++ interpreter you have to generate a dictionary for your classes.

The dictionary will be generated using the program **rootcint** that comes with the RDK (Root Development Kit). Below follow two examples of how to use rootcint to generate a dictionary and how to compile and link this dictionary with your classes and the ROOT libraries.

Dictionary Generation for Interactive Access Without I/O and RTTI

The first example shows how a stand alone class, i.e. a class not inheriting from ROOT's `TObject` class can be made accessible to the interpreter for interactive manipulation.

We begin with a simple header file `MyClass.h` defining `class MyClass`:

```
//--------------------------------------------------
#ifndef __MyClass__
#define __MyClass__
class MyClass {
private:
  float   fX;     //x position in centimeters
  float   fY;     //y position in centimeters
public:
  MyClass();
  void    Print() const;
  void    SetX(float x) { fX = x; }
  void    SetY(float y) { fY = y; }
};
#endif
//--------------------------------------------------
```

And its implementation file `MyClass.C`:

```
//--------------------------------------------------
#include <iostream.h>
#include "MyClass.h"
MyClass::MyClass()
{
  fX = -1;
  fY = -1;
}
void MyClass::Print() const
{
  cout << "fX = " << fX << ", fY = " << fY << endl;
}
//--------------------------------------------------
```

To make this class accessible via the command line we need to link it with a small ROOT main program, `main.C`, that creates and calls the command line interpreter:

```
//--------------------------------------------------
#include "TROOT.h"
#include "TRint.h"
int Error; //left undefined by Motif
extern void  InitGui();  // initializer for GUI needed for interactive interface
VoidFuncPtr_t initfuncs[] = { InitGui, 0 };
// Initialize the ROOT system
TROOT root("Rint","The ROOT Interactive Interface", initfuncs);
int main(int argc, char **argv)
{
   // Create interactive interface
   TRint *theApp = new TRint("ROOT example", &argc, argv, NULL, 0);
   // Run interactive interface
   theApp->Run();
   return(0);
}
//--------------------------------------------------
```

Using the following `Makefile` we can use `make` to build the program `myroot` that will give the user access to class `MyClass` via the C++ interpreter:

```
#--------------------------------------------------
CXXFLAGS      = -w -g +a1 -I$(ROOTSYS)/include
LDFLAGS       = -w -g +a1
LD            = CC
```

```
LIBS          = $(ROOTSYS)/lib/*.sl -lXm -lXt -lX11 -lm -lPW -ldld
HDRS          = MyClass.h
SRCS          = main.C MyClass.C mydict.C
OBJS          = main.o MyClass.o mydict.o
PROGRAM       = myroot
all:            $(PROGRAM)
$(PROGRAM):     $(OBJS)
               @echo "Linking $(PROGRAM) ..."
               @$(LD) $(LDFLAGS) $(OBJS) $(LIBS) -o $(PROGRAM)
               @echo "done"
clean:;         @rm -f $(OBJS) core
###
MyClass.o: MyClass.h
mydict.C: MyClass.h
       @echo "Generating dictionary ..."
       @rootcint mydict.C -c MyClass.h
```

#---

This `Makefile` assumes that the environment variable `$ROOTSYS` has been correctly set (as described in the `AA_README` file that comes with the RDK) and that `$ROOTSYS/bin` has been added to `$PATH`. The compiler options are for HP-UX (for options for other platforms see the compile scripts in the `root/test` directory that comes with the RDK).

The line:

rootcint mydict.C -c MyClass.h

shows how **rootcint** is used to generate the dictionary files `mydict.h` and `mydict.C`. To get a full list and a description of all command line options supported by rootcint do:

rootcint -?

To see how to run `myroot` and to create and manipulate `MyClass` objects via the interpreter see: "CINT as Command Line and Macro Interpreter".

Dictionary Generation for Interactive Access Including I/O and RTTI

In this example we'll show how to fully integrate a set of classes into the ROOT system and how a shared library of these classes can be dynamically loaded into a running ROOTsession. We'll create an Event class which contains a list of Tracks. Here are the files `Event.h` and `Track.h`:

```
//---------------------------------------------------
#ifndef __Event__
#define __Event__
#include "TObject.h"
class TCollection;
class Track;
class Event : public TObject {
private:
  Int_t         fId;                //event sequential id
  Float_t       fTotalMom;          //total momentum
  TCollection   *fTracks;           //collection of tracks
public:
  Event() { fId = 0; fTracks = 0; }
  Event(Int_t id);
  ~Event();
  void    AddTrack(Track *t);
  Int_t   GetId() const { return fId; }
  Int_t   GetNoTracks() const;
  void    Print(Option_t *opt="");
  Float_t TotalMomentum();
  ClassDef(Event,1)  //Simple event class
};
#endif
```

//---

```
//---------------------------------------------------
#ifndef __Track__
#define __Track__
#include "TObject.h"
class Event;
class Track : public TObject {
private:
  Int_t    fId;       //track sequential id
  Event   *fEvent;    //event to which track belongs
  Float_t  fPx;       //x part of track momentum
  Float_t  fPy;       //y part of track momentum
  Float_t  fPz;       //z part of track momentum
public:
```

```
  Track() { fId = 0; fEvent = 0; fPx = fPy = fPz = 0; }
  Track(Int_t id, Event *ev, Float_t px, Float_t py, Float_t pz);
  Float_t  Momentum() const;
  Event   *GetEvent() const { return fEvent; }
  void     Print(Option_t *opt="");
  ClassDef(Track,1)  //Simple track class
};
#endif
```

//---

The first things to notice in these header files are the usage of the `ClassDef` macro and the default contructors of the `Event` and `Track` classes. Also notice the usage of comments to describe the data members and the comment after the `ClassDef` macro to describe the class. The intended usage of these classes is that one creates and event object with a certain id and then add tracks to the event. As one can see the track objects contain a pointer to the event to which they belong. This to show that the I/O system will correctly handle circular references. As an aside, note that although both header files contain references to each others objects there is no need to include the complete header files. A simple class declaration is enough ("`class Track;`"). This does not seem important now, but when a system grows it can save a lot of time during compilation when not every file that includes `Event.h` forces also the reading of `Track.h` or vice versa.

Next the implementation of these two classes. `Event.C`:

```
//---------------------------------------------------
#include <iostream.h>
#include "TOrdCollection.h"
#include "Event.h"
#include "Track.h"
ClassImp(Event)
Event::Event(Int_t id)
{
  fId = id;
  fTracks = new TOrdCollection;
}
Event::~Event()
{
  delete fTracks;
}
void Event::AddTrack(Track *t)
{
  fTracks->Add(t);
}
Int_t Event::GetNoTracks() const
{
  return fTracks->GetSize();
}
Float_t Event::TotalMomentum()
{
  TIter next(fTracks);
  Track *t;
  while (t = (Track *)next())
     fTotalMom += t->Momentum();
  return fTotalMom;
}
void Event::Print(Option_t *)
{
  cout << "*** Event=" << fId << " No of tracks=" << GetNoTracks() << endl;
  fTracks->Print();
}
```

//---

And `Track.C`:

```
//---------------------------------------------------
#include <iostream.h>
#include "TMath.h"
#include "Track.h"
#include "Event.h"
ClassImp(Track)
Track::Track(Int_t id, Event *ev, Float_t px, Float_t py, Float_t pz)
{
  fId    = id;
  fEvent = ev;
  fPx    = px;
  fPy    = py;
  fPz    = pz;
}
Float_t Track::Momentum() const
{
  return TMath::Sqrt(fPx*fPx+fPy*fPy+fPz*fPz);
}
void Track::Print(Option_t *)
```

```
{
  cout << "id=" << fId << " event#=" << fEvent->GetId() << " px=" << fPx
       << " py=" << fPy << " pz=" << fPz << endl;
}
```

//--

In the implementation files we notice the `ClassImp` macro's. Further, in `Event.C`, we see how we create a container class, `TOrdCollection`, and how we iterate over the collection using a `TIter` object. Note also how in `Event.h` we did not specify what kind of container we were going to use. Since all containers inherit from `TCollection` we are free to choose in the implementation file the collection with the right properties for the job. We don't have to change the header in case we want to use another container class.

To create the `event.sl` shared library on HP-UX we use the following `Makefile`:

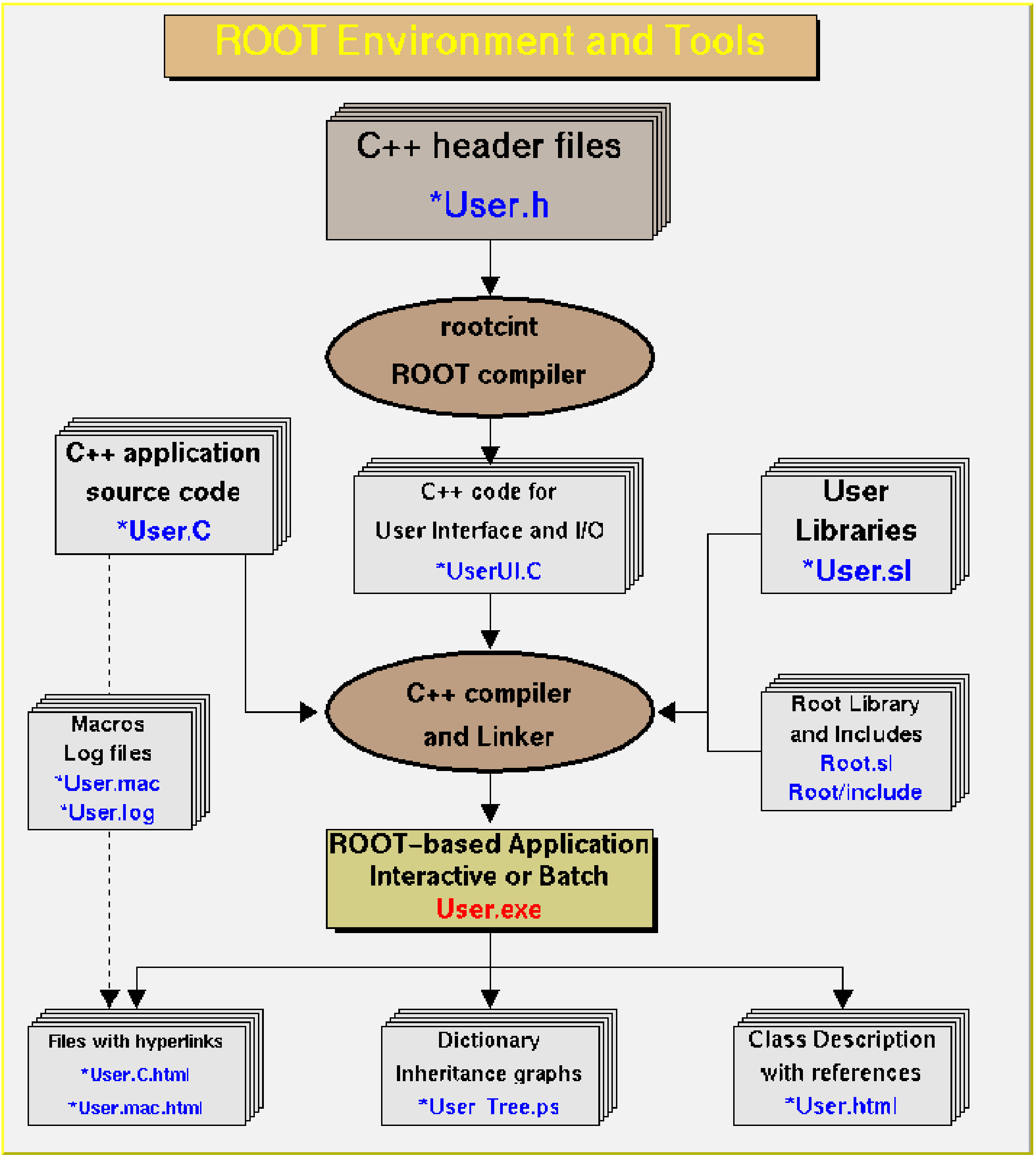
```
#----------------------------------------------------
CXXFLAGS      = -w -g +a1 +Z -I$(ROOTSYS)/include
LDFLAGS       = -w -g +a1 -b
LD            = CC
HDRS          = Event.h Track.h eventdict.h
SRCS          = Event.C Track.C eventdict.C
OBJS          = Event.o Track.o eventdict.o
PROGRAM       = event.sl
all:            $(PROGRAM)
$(PROGRAM):     $(OBJS)
               @echo "Linking $(PROGRAM) ..."
               @/bin/rm -f $(PROGRAM)
               @$(LD) $(LDFLAGS) $(OBJS) -o $(PROGRAM)
               @chmod 555 $(PROGRAM)
               @echo "done"
clean:;         @rm -f $(OBJS) core
###
Event.o: Event.h
Track.o: Track.h
eventdict.C: Event.h Track.h
       @echo "Generating dictionary ..."
       @rootcint eventdict.C -c Event.h Track.h
```

#--

After running `make` have a look at the file `eventdict.C`. At the bottom of this file we see the `TBuffer &operator>>()`, `Streamer()` and `ShowMembers()` methods for our two classes, all automatically generated by **rootcint**. This is all there is to get extensive RTTI and full ROOT object I/O. No need for seperate IDL files or meta header files describing the class structure.

To see how to load `event.sl` in a running ROOT process and how to create, write and read events and tracks see: "Extending ROOT with Shared Libraries and an Example of Object I/O"

The figure below depicts the different stages and files involved in the dictionary creation process described above.


ROOT Environment and Tools
C++ header files
*User.h
rootcint
ROOT compiler
C++ application
source code
*User.C
C++ code for
User Interface and I/O
*UserUI.C
User
Libraries
*User.sl
Macros
Log files
*User.mac
*User.log
C++ compiler
and Linker
Root Library
and Includes
Root.sl
Root/include
ROOT-based Application
Interactive or Batch
User.exe
Files with hyperlinks
*User.C.html
*User.mac.html
Dictionary
Inheritance graphs
*User_Tree.ps
Class Description
with references
*User.html

Adding Your Own Classes to ROOT

If you want to integrate your classes into the ROOT system, to enjoy features like, *extensive RTTI* and *ROOT object I/O*, you have to add the following line to your class header files (as part of the public part of the class definition):

ClassDef(*ClassName,ClassVersionID*) *//The class title*

See for example TLine.h. The **ClassVersionID** is used by the ROOT I/O system. It is written on the output stream and during reading you can check this version ID and take appropriate action depending on the value of the ID (see `Streamer`). Every time you change the layout of a class you should increase its **ClassVersionID** by one. The **ClassVersionID** should be **>=1**. Set **ClassVersionID=0** in case you don't need object I/O.

Similarly, in your implementation file you must add the statement:

ClassImp(ClassName)

See for example Line.C. Note that you *MUST* provide a default constructor for your classes, i.e. a constructor with zero parameters or with one or more parameters all with default values in case you want to use object I/O. If not you will get a compile time error.

The `ClassDef` and `ClassImp` macros are defined in the file Rtypes.h. This file is referenced by all ROOT include files, so you will automatically get them if you use a ROOT include file.

The `ClassDef` and `ClassImp` macros are necessary to link your classes to the dictionary generated by CINT to get access to the RTTI and object I/O features of ROOT. The RTTI system allows you to, a.o. find out to which class an object belongs, its baseclasses, its datamembers and methods, the method signatures, etc. This information is used to make advanced object browsers and by the automatic documentation generation system. The object I/O system allows you to store and retrieve objects (and arbitrarily complex object structures) from a ROOT database.

*If you are **only** interested in interactive access to your classes via the command line or macro processor you do not need to use the `ClassDef` and `ClassImp` macros.*

See "CINT as Dictionary Generator" for a description on how to generate a dictionary.

The CINT Interpreter Interface

CINT as embedded in ROOT can be used as command line interpreter and as macro processor, where macros are "small" (up to at least 60000 loc) C++ programs. Thanks to CINT the ROOT system can present the user a single language environment: C++ as implementation, macro and command line language.

The advantages of a single language model are clear. Especially when writing macros. Typically macros start as small prototypes that need frequent modification. While execution speed is not important a short edit-execute cycle is. However, once macros have grown to full programs and have become stable, the need for fast execution in production jobs becomes important. Thanks to the fact that the macros are in "standard" C++ we can simply compile and dynamically link the macros with the ROOT system and execute them at full speed.

First we will show how to use the ROOT/CINT command line interpreter and next how to execute C++ macros. Along the way we'll describe the very few extensions made to CINT's C++ to facilitate a smooth integration with ROOT.

CINT as Command Line Interpreter

Lets run the program `myroot` that we created in "CINT as Dictionary Generator" and see how we can create and manipulate `MyClass` objects via the interpreter (for brevity's sake some non essential output has been removed):

```
$ myroot
 *******************************************
 *                                         *
 *        W E L C O M E  to  R O O T       *
 *                                         *
 *   Version   0.90/12  18 January 1997    *
 *                                         *
 *  You are welcome to visit our Web site  *
 *          http://root.cern.ch            *
 *                                         *
 *******************************************
CINT/ROOT C/C++ Interpreter version 5.11.29, Dec 5 1996
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0] MyClass a
root [1] a.Print()
fX = -1, fY = -1
root [2] a.SetX(10)
root [3] a.SetY(11)
root [4] a.Print()
fX = 10, fY = 11
root [5] .g
...
...
0x4045c7f8 class MyClass a , size=8
 0x0        private: float fX //x position in centimeters
 0x0        private: float fY //y position in centimeters
root [6] .class MyClass
===========================================================================
class MyClass
size=0x8
(tagnum=438,voffset=-1,isabstract=0,parent=-1,gcomp=0,=~cd=0)
List of base class--------------------------------------------------------
List of member variable---------------------------------------------------
Defined in MyClass
0x0        private: float fX //x position in centimeters
0x0        private: float fY //y position in centimeters
List of member function---------------------------------------------------
Defined in MyClass
filename       line:size busy function type and name
(compiled)        0:0    0 public: class MyClass MyClass(void);
(compiled)        0:0    0 public: void Print(void);
(compiled)        0:0    0 public: void SetX(float x);
(compiled)        0:0    0 public: void SetY(float y);
(compiled)        0:0    0 public: class MyClass MyClass(class MyClass&);
(compiled)        0:0    0 public: void ~MyClass(void);
```

root [7] .q

In the above example we first create an object `a` of class `MyClass`. Next we call the `Print()` method (and notice that the constructor has been executed correctly), set some variables and call again the `Print()` method. Everything works as expected. Notice that when issueing single line commands we may omit the trailing "`;`".

Next we execute a "raw" interpreter command `.g` to list all global symbols currently defined in the interpreter followed by the command `.class` *classname* to show the definition of `MyClass` as known to the interpreter. We exit the session by typing `.q`. Here we notice that all interpreter commands start with a "`.`", except for the "`?`" command which you can use to get a list of all available interpreter commands.

The ROOT command line interface supports emacs style command line editing. Also the command history is saved between sessions.

Now lets execute a multi line command. We start again `myroot` and type:

```
root [0] {
end with '}'> MyClass a;
end with '}'> for (int i = 0; i < 5; i++) {
end with '}'>    a.SetX(i);
end with '}'>    a.SetY(i+1);
end with '}'>    a.Print();
end with '}'> }
end with '}'> }
fX = 0, fY = 1
fX = 1, fY = 2
fX = 2, fY = 3
fX = 3, fY = 4
fX = 4, fY = 5
```

root [1] .q

A multi line command starts always with a "`{`" and ends with a "`}`". As long as your are in multi line input mode you are prompted with "`end with '}'>`". After typing the closing "`}`" the command will be executed. Notice that now you have to terminate every line, like in C++, with a "`;`". Inserting many lines in this way requires typing without making any errors. There is no way to correct a once entered line. Therefore it is much easier to put such a sequence of commands in a macro file.

CINT as Macro Processor

CINT macro files contain pure C++ code. They can contain a simple sequence of statements like in the multi command line example given above, but also arbitrarily complex class and function definitions.

Lets start with a macro containing a simple list of statements (like the multi command line example given in the previous section). This type of macro must start with a "`{`" and end with a "`}`". Assume the file is called `macro1.C`:

```
//--------------------------------------------------
{
#include <iostream.h>
  cout << " Hello" << endl;
  float x = 3.;
  float y = 5.;
  int   i = 101;
  cout << " x = " << x << " y = " << y << " i = " << i << endl;
  return 0;
}
```

//--

To execute the stream of statements in `macro1.C` do:

```
root [1] .x macro1.C       // loads the contents of file macro1.C and
                          // executes all statements in the global scope
```

One can re-execute the statements by re-issueing "`.x macro1.C`" (since there is no function entry point).

Now copy file `macro1.C` to `macro2.C` and add a function statement. Like this:

```
//--------------------------------------------------
#include <iostream.h>
int main()
{
  cout << " Hello" << endl;
  float x = 3.;
  float y = 5.;
  int   i= 101;
  cout << " x = " << x << " y = " << y << " i = " << i << endl;
  return 0;
}
```

//--

Notice that no surrounding "`{`" and "`}`" are required in this case.

To execute function `main()` in `macro2.C` do:

```
root [1] .L macro2.C        // load the contents of file macro2.C in memory
root [2] main()             // execute entry point main()
Hello
x = 3 y = 5 i = 101
(int)0
root [3] main()             // execute main() again
Hello
x = 3 y = 5 i = 101
(int)0
root [4] .func              // list all functions known by CINT
filename       line:size busy function type and name
...
...
```

macro2.C 4:9 0 public: int main();

The last command shows that `main()` has been loaded from file `macro2.C`, that the function `main()` starts on line 4 and is 9 lines long. Notice that once a function has been loaded it becomes part of the system just like a compiled function.

Now we copy file `macro2.C` to `macro3.C` and change the function name from `main()` to `macro3(int j = 10)`:

```
//--------------------------------------------------
#include <iostream.h>
int macro3(int j = 10)
{
  cout << " Hello" << endl;
  float x = 3.;
  float y = 5.;
  int   i = j;
  cout << " x = " << x << " y = " << y << " i = " << i << endl;
  return 0;
}
```

//--

To execute `macro3()` in `macro3.C` type:

```
root [1] .x macro3.C(8)    // loads the contents of file macro3.C and
```

// executes entry point macro3(8)

Note that the above only works when the filename (minus extension) and function entry point are both the same. This shortcut is mainly interesting to run large one-off macros where one wants to pass arguments at the same time (instead of doing the "`.L`" step first). Function `macro3()` can still be executed multiple times:

```
root [2] macro3()
Hello
x = 3 y = 5 i = 10
(int)0
root [3] macro3(33)
Hello
x = 3 y = 5 i = 33
```

(int)0

A Macro Containing a Class Definition

As an example of a macro containing class definitions we take the small class `MyClass` that we used in the " CINT as Dictionary Generator" chapter and we introduce a subclass `Child`:

```
//--------------------------------------------------
#include <iostream.h>
class MyClass {
private:
  float   fX;     //x position in centimeters
  float   fY;     //y position in centimeters
public:
  MyClass() { fX = fY = -1; }
  virtual void Print() const;
  void         SetX(float x) { fX = x; }
  void         SetY(float y) { fY = y; }
};
void MyClass::Print() const
{
  cout << "fX = " << fX << ", fY = " << fY << endl;
}
class Child : public MyClass {
public:
  void Print() const;
};
void Child::Print() const
{
```

```
  cout << "This is Child::Print()" << endl;
  MyClass::Print();
}
```

//--

The only changes we made was to put `MyClass.h` and the `MyClass.C` files together in file `macro4.C` and add the `Child` class. To make things a little less trivial we made `MyClass::Print()` virtual and overrode it in the `Child` class.

To execute `macro4.C` do:

```
root [0] .L macro4.C
root [1] MyClass *a = new Child
root [2] a.Print()
This is Child::Print()
fX = -1, fY = -1
root [3] a.SetX(10)
root [4] a.SetY(12)
root [5] a.Print()
This is Child::Print()
fX = 10, fY = 12
root [6] .class MyClass
===========================================================================
class MyClass
size=0x8 FILE:macro4.C LINE:3
(tagnum=459,voffset=-1,isabstract=0,parent=-1,gcomp=0,=~cd=1)
List of base class--------------------------------------------------------
List of member variable---------------------------------------------------
Defined in MyClass
0x0        private: float fX
0x4        private: float fY
List of member function---------------------------------------------------
Defined in MyClass
filename       line:size busy function type and name
macro4.C         16:5    0 public: class MyClass MyClass(void);
macro4.C         22:4    0 public: void Print(void);
macro4.C         12:1    0 public: void SetX(float x);
macro4.C         13:1    0 public: void SetY(float y);
```

root [7] .q

As you can see an interpreted class behaves just like a compiled class (e.g. like `MyClass` in `myroot`).

Reseting the Interpreter Environment

Loading files and executing functions will in general result in many objects being created on the interpreter's internal stack. Once in a while it might be necessary to reset the interpreter environment. There are basically two way to reset the environment:

- Using the interpreter command `.reset` or
- Using the ROOT function `gROOT.Reset()`

The first method clears the complete interpreter environment, while the second method clears the environment to the status just before executing the last macro. Therefore multi command line macros often start with the statement `gROOT.Reset()` (where `gROOT` is a pointer to the global ROOT object, see also `TROOT`).

Debugging Macros

A powerful feature of CINT is the ablity to debug interpreted methods and functions by means of setting breakpoints and being able to single step through the code and print variable values on the way. Assume we have `macro4.C` still loaded, we can then do:

```
root [1] .b Child::Print
Break point set to line 26 macro4.C
root [2] a.Print()
26   Child::Print() const
27   {
28      cout << "This is Child::Print()" <<endl;
FILE:macro4.C LINE:28 cint> .s
311  operator<<(ostream& ostr,G__CINT_ENDL& i) {return(endl(ostr));
FILE:iostream.h LINE:311 cint> .s
}
This is Child::Print()
29      MyClass::Print();
FILE:macro4.C LINE:29 cint> .s
16   MyClass::Print() const
17   {
18      cout << "fX = " << fX << ", fY = " << fY << endl;
FILE:macro4.C LINE:18 cint> .p fX
```

```
(float)1.000000000000e+01
FILE:macro4.C LINE:18 cint> .s
311  operator<<(ostream& ostr,G__CINT_ENDL& i) {return(endl(ostr));
FILE:iostream.h LINE:311 cint> .s
}
fX = 10, fY = 12
19   }
30   }
2    }
```

root [3] .q

In the above debug session we first set a breakpoint on the `Child::Print()` method. Then we execute the method. When we hit the breakpoint CINT returns a prompt and we can issue any of the debug commands (for a full list of debug commands, type a `?` at the ROOT prompt). In the above example we single step till we are in `MyClass::Print()` where we examine the value of `fX`. We can only set breakpoints on or in functions, therefore macros without function statement can not be debugged (like simple multi command line macros). In such cases just wrap the sequence of statements in a function.

ROOT/CINT Extensions to C++

In the next example we demonstrate three of the most important extensions ROOT/CINT makes to C++. Start basic `root` or `myroot` in the directory `root/tutorials` that comes with the RDK (make sure the file `hsimple.root` is there):

```
root [1] f = new TFile("hsimple.root")
(class TFile*)0x4045e690
root [2] f.ls()
TFile**         hsimple.root
TFile*         hsimple.root
 KEY: TH1F     hpx;1   This is the px distribution
 KEY: TH2F     hpxpy;1 py ps px
 KEY: THProfile        hprof;1 Profile of pz versus px
 KEY: TNtuple  ntuple;1        Demo ntuple
root [3] hpx.Draw()
NULL
Warning in <MakeDefCanvas>: creating a default canvas with name c1
```

root [4] .q

The first command shows the first extension; the declaration of "`f`" may be omitted when "`new`" is used. CINT will correctly create "`f`" as object of class "`TFile`".

The second extension is shown in the second command. Although "`f`" is a pointer to "`TFile`" we don't have to use the pointer dereferencing syntax "`->`" but can use the simple "." notation (since every object created by CINT is a heap object so CINT does not need the distinction between heap and stack based objects).

The third extension is more important. In case CINT can not find an object being referenced it will ask ROOT to search for an object with an identical name in the search path defined by `TROOT::FindObject()`. If ROOT finds the object it returns CINT a pointer to this object and a pointer to its class definition and CINT will execute the requested member function. This shortcut is quite natural for an interactive system and saves a lot of typing, e.g.:

```
root [4] TH1 *hpx = (TH1*)gROOT.FindObject("hpx")
```

root [5] hpx.Draw()

Of course when writing large macros, it is best to stay away from these shortcuts since otherwise you will later have problems compiling your macros using a real C++ compiler.

CINT Commands

Below a list of all "raw" CINT commands. To execute any of these commands they need to be preceded by a `.` (dot). The commands are case sensitive, i.e. `.L` is not the same a `.l`. To get this list while being in ROOT type a `?` at the command line prompt.

```
Dump:        n [file]              : create new readline dumpfile and start dump
            y [file]              : append readline dump to [file]
            z                     : stop readline dump
            < [file]              : execute readline dumpfile
            > [file]              : output redirect to [file]
            2> [file]             : error redirect to [file]
Help:        ?                     : help
            help                  : help
            /[keyword]            : help information for keyword
Completion:  [nam][Tab]            : complete symbol name start with [nam]
            [nam][Tab][Tab]       : list up all symbol name start with [nam]]
Shell:       ![shell]              : execute shell command
Source:      v <[line]>            : view source code <around [line]>
            V [stack]             : view source code in function call stack
            t                     : show function call stack
            f [file]              : view source file [file]
            trace <classname>     : turn on trace mode for class
            deltrace <classname>  : turn off trace mode for class
            break [classname]     : break at the memberfunc
            delbreak [classname]  : turn off memberfunc break
Evaluation:  p [expr]              : evaluate expression (no declaration/loop/condition)
            {[statements]}        : evaluate statement (any kind)
            x [file]              : load [file] and evaluate {statements} in the file
            X [file]              : load [file] and execute function [file](minus extension)
            E <[file]>            : open editor and evaluate {statements} in the file
Load/Unload: L [file]              : load [file]
            U [file]              : unload [file]
            reset                 : reset interpreter environment
Monitor:     g <[var]>             : list global variable
            l <[var]>             : list local variable
            class <[name]>        : show class definition (one level)
            Class <[name]>        : show class definition (all level)
            typedef <name>        : show typedefs
            function              : show interpreted functions
            macro                 : show macro functions
            template              : show templates
            include               : show include paths
            file                  : show loaded files
            garbage               : show garbage collection buffer
            Garbage               : Do garbage collection
            cover [file]          : save trace coverage
            return [val]          : return undefined symbol value
Run:         S                     : step over function/loop
            s                     : step into function/loop
            i                     : ignore and step over
            c <[line]>            : continue <to [line]>
            e                     : step out from function
            b [line]              : set break point
            db [line]             : delete break point
            a [assert]            : break only if assertion is true
Quit:        q                     : quit cint execution
            qqq                   : quit anyway
```

Emacs Style Command Line Editing

Entering printable keys generally inserts new text into the buffer (unless in overwrite mode, see below). Other special keys can be used to modify the text in the buffer. In the description of the keys below, `^n` means `Control-n`, or holding the CONTROL key down while pressing "n". Errors will ring the terminal bell.

```
^A/^E   : Move cursor to beginning/end of the line.
^F/^B   : Move cursor forward/backward one character.
^D      : Delete the character under the cursor.
^H, DEL : Delete the character to the left of the cursor.
^K      : Kill from the cursor to the end of line.
^L      : Redraw current line.
^O      : Toggle overwrite/insert mode. Initially in insert mode. Text
          added in overwrite mode (including yanks) overwrite
          existing text, while insert mode does not overwrite.
^P/^N   : Move to previous/next item on history list.
^R/^S   : Perform incremental reverse/forward search for string on
          the history list.  Typing normal characters adds to the current
          search string and searches for a match. Typing ^R/^S marks
          the start of a new search, and moves on to the next match.
          Typing ^H or DEL deletes the last character from the search
          string, and searches from the starting location of the last search.
          Therefore, repeated DEL's appear to unwind to the match nearest
          the point at which the last ^R or ^S was typed.  If DEL is
          repeated until the search string is empty the search location
          begins from the start of the history list.  Typing ESC or
          any other editing character accepts the current match and
          loads it into the buffer, terminating the search.
^T      : Toggle the characters under and to the left of the cursor.
^U      : Kill from beginning to the end of the line.
^Y      : Yank previously killed text back at current location.  Note that
          this will overwrite or insert, depending on the current mode.
TAB     : By default adds spaces to buffer to get to next TAB stop
          (just after every 8th column), although this may be rebound by the
          programmer, as described below.
NL, CR  : returns current buffer to the program.
```

DOS and ANSI terminal arrow key sequences are recognized, and act like:

```
 up    : same as ^P
 down  : same as ^N
 left  : same as ^B
 right : same as ^F
```

ROOT Tutorials

1. Starting an Interactive Session
2. The Hello World Example
3. Using CINT as Command Line and Macro Interpreter
4. An Example of Object Oriented User Interface
5. Reading an ascii file and making an ntuple
6. An Example Creating Histograms/Ntuples on a File
7. Filling Several Histograms and Some Graphics Options
8. Simple Formula and Functions
9. Histograms with Random Numbers from a Function
10. A Simple Fitting Example
11. Using Minuit to fit non-equidistant points
12. How to fit in a sub-range of an histogram
13. Fitting with a user-defined function
14. Drawing Options for 1D Histograms
15. Example of a Simple Graph with axis titles
16. Examples of a Graph with error bars
17. Surfaces drawing options
18. Examples of 3-D Polymarkers
19. The Geometry Shapes and the X3D viewer
20. Creating and Viewing Geometries
21. Ntuples and Selections
22. Creation of a ROOT Tree (program)
23. Reading all events of a ROOT Tree (macro)
24. Reading selected events of a ROOT Tree (macro)
25. Example of analysis code with Chains (macro)
26. Example of Minimization (program)
27. Example of a Menu Controlbar
28. Using the Collection Classes
29. Adding Your Own Classes to ROOT
30. Using CINT as Dictionary Generator
31. Example of Client-Server communication: the Client
32. Example of Client-Server communication: the Server
33. Example of Shared memory: producer
34. Example of Shared memory: consumer
35. Extending ROOT with Shared Libraries and an Example of Object I/O
36. Automatic HTML Document Generation
37. Object Browser and Inspector

To run the tutorials yourself download the ROOT Development Kit (RDK).

The examples have been designed to run on a color workstation. Opaque moving of objects in the canvas requires a machine with a reasonable response time. If you run these examples on a slow machine or saturated time sharing system, you can remove the line **Rint.Canvas.MoveOpaque: true** from the file **.rootrc** to improve dragging performance.

The ROOT Object I/O System

Why Not Use an OO Data Base Instead?

The ROOT system includes a set of classes to support Input/Output from/to machine independent files. The system is designed to be particularly efficient for objects frequently manipulated by physicists: histograms, ntuples, trees and events.

One could argue that this functionality may also be provided by Object Oriented Data Base Management Systems (OODBMS). We consider OODBMS's as potential candidates for the replacement of tools like **HEPDB** or **FATMEN**, i.e. when locking and concurrent writing is required. But we do not believe that they provide a solution for the types of objects mentioned above. Why?

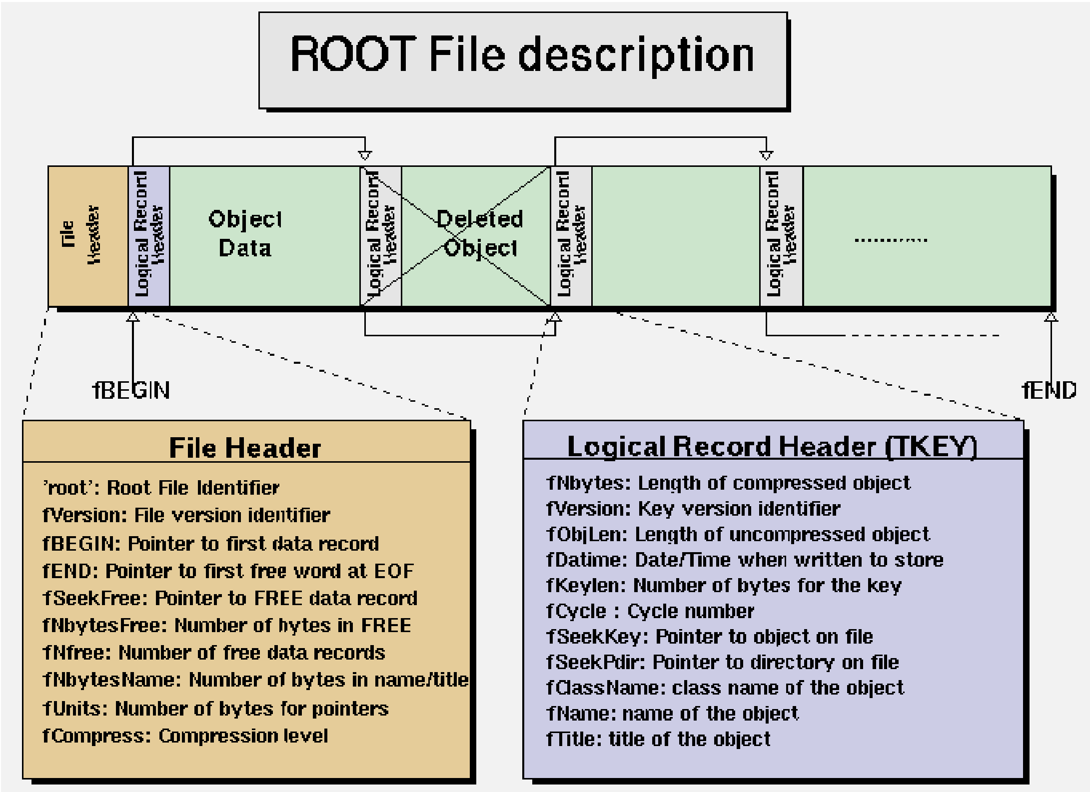
- Interactive computing is towards commodity desktop and notebook devices. They will be heavily used for histogram manipulation and data presentation. This should not require a special connection to a central data base or a license server (think of home computing).
- OODBMS's, by definition, are designed to store complete objects. Data clustering is organized around objects and containers of objects. They are not designed to access only a subset of the object attributes. We have demonstrated with the **PAW** column-wise ntuples the usefulness of having access to single attributes. The ROOT Tree functionality cannot be provided in an efficient way by the OODBMS model.
- OO data bases do not support on the fly data compression. We are designing experiments that will generate massive amounts of data. The cost of direct access devices for tens of terabytes may be a dominant factor in the cost of computing.
- Attribute range specification is not supported. A 4 byte integer cannot be saved as a single byte.
- The data bases companies are small and fragile. Will they survive in a few years? The technology is not yet mature and compatibility between vendors is not guaranteed.
- We prefer not to discuss the question of the cost and the manpower resources necessary to support a commercial data base.

If a connection to an OODBMS is required, we understand now how to make an automatic interface between the ROOT dictionary and the data base schema definition language (ddl). To use an OO data base, the user classes must inherit from a base class **ooObj**. This inheritance affects many places in the user's code (casting operations in iterators, if templates are not used). The ROOT base class `TObject` could be used instead. A special version of `TObject` could be provided to make an automatic and transparent interface to the ooObj class.

The Physical File Structure

The **ROOT** file structure is shown in the figure below. A file has a file header (less than 64 bytes) and then several logical records of variable length. The first 4 bytes of each physical record are an integer holding the number of bytes in this record. If the number of bytes is negative, it identifies a deleted record that can be reused in a subsequent write operation. The following bytes (documented on the picture) contain all the information to uniquely identify a data block on the file.

The TFile::Map member function can be used to view the contents of a file by reading sequentially all the data blocks. The information stored by the **ROOT** Output functions is always in machine independent format (ASCII, IEEE floating point, Big Endian byte ordering). The redundancy in the logical record header can be used in case of file corruption or disk errors, to rebuild the original structure. Data in the logical records can be compressed or uncompressed, but the logical record header is never compressed. The first logical record on a file always contain the description of the top level directory of the file.



The logical records contain the following possible data:

• File/directory information.

• A standard Object as written by the TObject::Write function. When the Write function is called,a `TKey` object is created. This TKey object is the logical record header.

• A `TTree` buffer. When a `TTree` buffer is written, a `TBasket` object is created. `TBasket` is derived from `TKey` and contains additional information specific to the `TTree` navigation logic.

• A user buffer. In the same way as for `TTree` objects, the user has the possibility to define a new class derived from `TKey` and optimize it for his objects.

Note that the concept of record length or block size has completely disappeared from the **ROOT** terminology. This simplifies considerably the logic of the system. It makes also simpler the implementation of memory mapping techniques.

The Logical File Structure

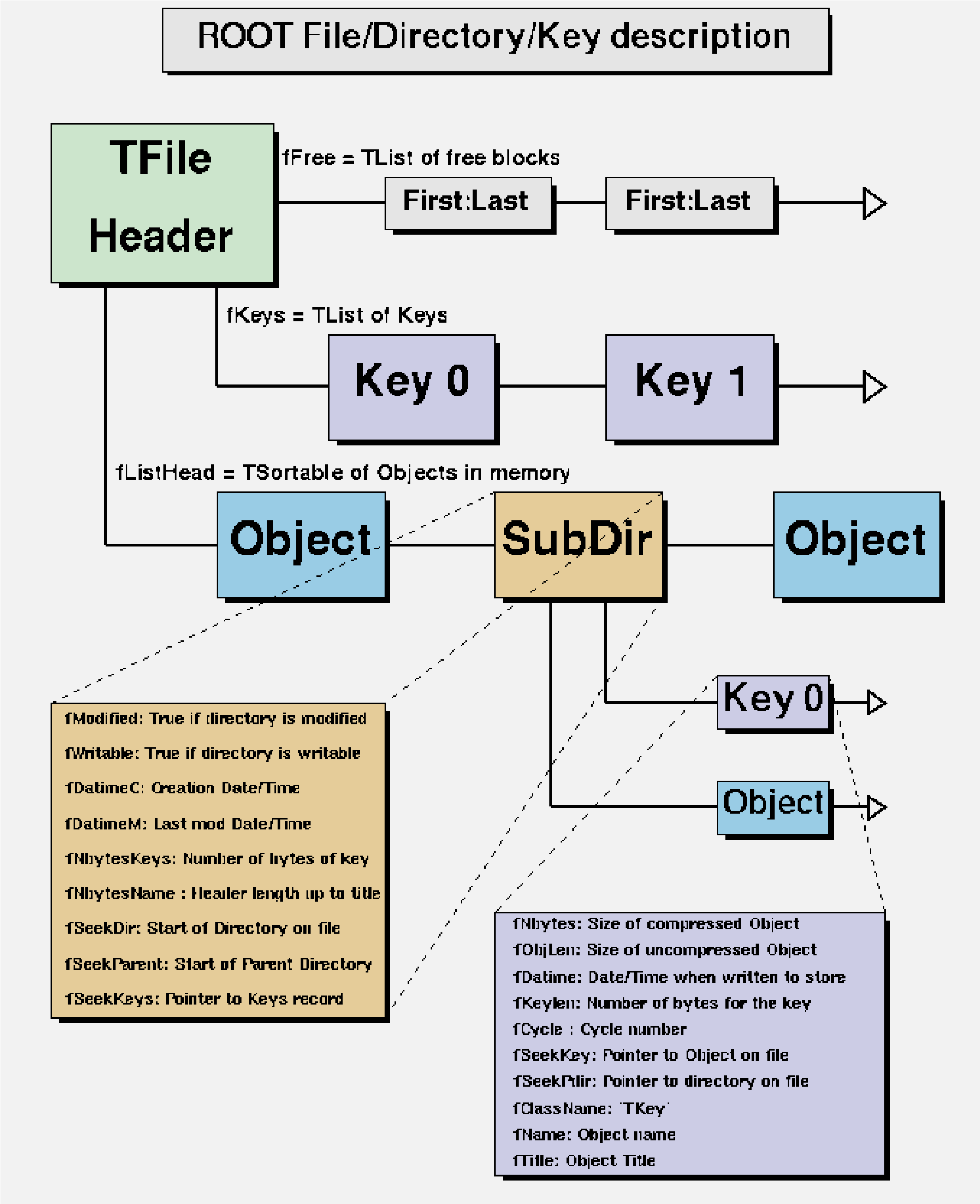
A **ROOT** file is like a Unix file system. It can contain directories and objects with an unlimited number of levels. Each directory has an associated list of **keys** kept in memory until the file is closed. Finding an object on the file is done in two steps:

• Find the position of the key object using the key name.

• Get the position of the object on the file (fSeekKey).

ROOT objects are always written in consecutive order on the file. By default, the directory description is written when the `TDirectory` constructor is invoked. However, if in a previous session, objects were deleted, the space released can be used by newly created objects. In this case, **ROOT** tries to find the best free block. The list of free blocks is a list supported by the `TFile` object. If a free block has a length that matches the length of the new stored object, the object is written in the free block and this free block is deleted from the list of free blocks. If not, the first free block bigger than the object is used. When a file is closed, the linked list of each directory is written to the file.

Thanks to this structure, a **ROOT** file can be read sequentially if all objects must be processed, or individual objects can be randomly accessed using the information in the `TKey` object. The `TTree` objects use a variant of the standard `TKeys(TBasket)`. The `TBasket` keys are designed to address randomly a large quantity of objects (`TBranch` buffers) in very large files.



Support for Class/Schema Evolution

A **ROOT** file will in general be written with the same version of a class library. However, in the life time of a collaboration, the definition of many classes is likely to change frequently.

The problem is made even more complicated by the use of inheritance. Assume a class D with its base classes C, B and A. An object of class D must be identified by the four version numbers of its composing classes.

We have implemented in the **ROOT** files a stamping mechanism that guarantees that old files can always be read by new libraries. A likely case is the one of an analysis program looping on many data sets generated across the years with different class definitions.

The header files containing the class definition must include the macro:

`ClassDef(Classname,VersionID)`, e.g. `ClassDef(TLine,1)`

The `ClassDef` macro is defined in the `Rtypes.h` include file which is automatically referenced by any include file using `TObject` derived classes. The second parameter of `ClassDef` is an integer representing the class version number. When the file is processed by the CINT Dictionary Generator the version information is saved in the dictionary data structure and is later available at execution time in the `Streamer()` I/O method.

The Input/Output Functions

An object is written to a file via the `TObject::Write()` function. This operation consists of the following steps:

- Creation of a support `TKey` object in the current directory.
- The `TKey` object creates a `TBuffer` object.
- The `TBuffer` object is filled via the `Streamer()` function of a class.
- If the file is compressed (default) a second buffer is created to hold the compressed buffer.
- Reservation of the corresponding space in the file by looking in the `TFree` list of free blocks of the file.
- The buffer is written to the file.

An object is read from the file into memory via the `TKey::Read()` function using the following sequence of operations:

- Search the key address by name in the current directory. The list of all the key names is contained in a hash-list.
- Create the buffer(s) necessary to read the object from file.
- The `TKey` object includes the name of the class of the object on the file. Using the class name, the pointer to the class definition (`TClass`) is obtained by looking into the linked list of class names supported by the top level **gROOT** object. The `TClass::New()` function is called. This function invokes the default constructor of the class.
- The object's `Streamer()` member function is called.

The prototype for the `Streamer()` function is automatically declared by the `ClassDef` macro. An Example of code for the `Streamer()` function is shown below for the **ROOT** class `TShape`. This code illustrates how the `Streamer()` function deals with base classes, including multiple inheritance. The two data members `fNumber` and `fVisibility` are integers. `fMaterial` is a pointer to the material definition for this shape. In the likely case that many different shapes will reference the same material, only one copy of the referenced material will be written. The code for this function is automatically generated by the CINT Dictionary Generator.

```
//______________________________________________________________________
void TShape::Streamer(TBuffer &b)
{
   // Stream a TShape object
   if (b.IsReading()) {
     Version_t v = b.ReadVersion();
     TNamed::Streamer(b);
     TAttLine::Streamer(b);
     TAttFill::Streamer(b);
     b >> fNumber;
     b >> fVisibility;
     b >> fMaterial;
   } else {
     b.WriteVersion(TShape::IsA());
     TNamed::Streamer(b);
     TAttLine::Streamer(b);
     TAttFill::Streamer(b);
     b << fNumber;
     b << fVisibility;
     b << fMaterial;
   }
}
```

Compression or No Compression?

By default, objects are compressed before being written to a file. The **ROOT** compression algorithm is based on derivatives of the well known **gzip** algorithm.

This algorithm supports up to 9 levels of compression. The default compression level is 1. This level is specified as a parameter in the `TFile` constructor or can be modified by the `TFile::SetCompressionLevel()` function. If the level is set to 0, no compression is done. The performance of the compression algorithm can be seen on an object by object basis by using the `TFile::Map()` function. Experience with this algorithm tends to indicate a compression factor of 1.3 for raw data files and around 2 on most DSTs files.

Our choice of default compression level 1 is a compromise between time and compression efficiency. To get an idea of the **ROOT** compression capabilities, you can try the

gzip -1 file

on any of your files.

The time to uncompress a buffer is small compared to the compression time and is independent of the selected compression level. A **ROOT** file may contain objects written with different compression levels.

The Compression Algorithm

The algorithm identifies new text as repetitions of old text within a fixed-length sliding window trailing behind the new text.

The **deflation** process depends on being able to identify portions of the input text which are identical to earlier input (within a sliding window trailing behind the input currently being processed).

The most straightforward technique turns out to be the fastest for most cases: try all possible matches and select the longest. The key feature of this algorithm is that insertions into the string dictionary are very simple and thus fast, and deletions are avoided completely. Insertions are performed at each input character, whereas string matches are performed only when the previous match ends. So it is preferable to spend more time in matches to allow very fast string insertions and avoid deletions. The matching algorithm for small strings is inspired by that of Rabin & Karp. A brute force approach is used to find longer strings when a small match has been found. A similar algorithm is used in comic (by Jan-Mark Wams) and freeze (by Leonid Broukhis). A description of the Rabin and Karp algorithm is given in the book

"Algorithms" by R. Sedgewick, Addison-Wesley, p252.

see also

```
Fiala,E.R., and Greene,D.H.
```

Data Compression with Finite Windows, Comm.ACM, 32,4 (1989) 490-595

We would like to thank **Evgueni Tcherniaev** for providing a very recent version of the algorithm. When you are interested in more information you are invited to read the many comments and the copyright notice in the code (directory ZIP).

The ROOT Trees

For many years, the data flow model in HEP has been:

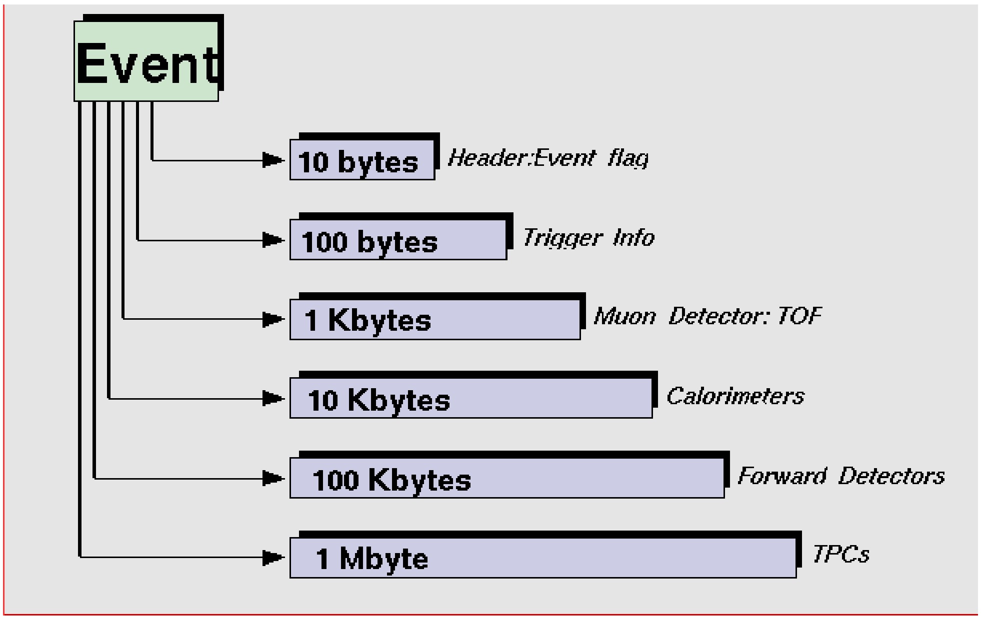
Raw Data Tapes --> Data Summary Tapes --> Mini/Micro DST's

The introduction of **Ntuples** in the **PAW** framework has proven to be very successful. Many experiments are using Ntuples as a convenient replacement for mini-DSTs or even DSTs.

The **PAW** Ntuples, however, were restricted to very simple data objects, collection of single variables or arrays.

With **ROOT**, we are introducing a new concept that we call **Trees**. **Trees provide the functionality of the Ntuples and much more**. The **Tree** architecture extends the concept of the Ntuple to all complex objects or data structures found in Raw Data tapes and DST's. The idea is that the same data model, same language, same style of queries can be used to all data sets in one experiment. **Trees** are designed to support not only complex objects, but also a very large number of them in a large number of files. The storage model using standard **ROOT** files is shown below.

In a conventional DST, all data structures of one event were written in a contiguous area on the file. This model has been very successful and robust for sequential files and when the analysis program requires access to a large number of all attributes of one event. On the other hand, this model was particularly inefficient when one had to iterate on a subset of the events or when a small subset of the event attributes was used. A typical structure for one event in HEP experiments is shown below. The pseudo-log scale used is just to indicate the large differences in object sizes from the various detectors.

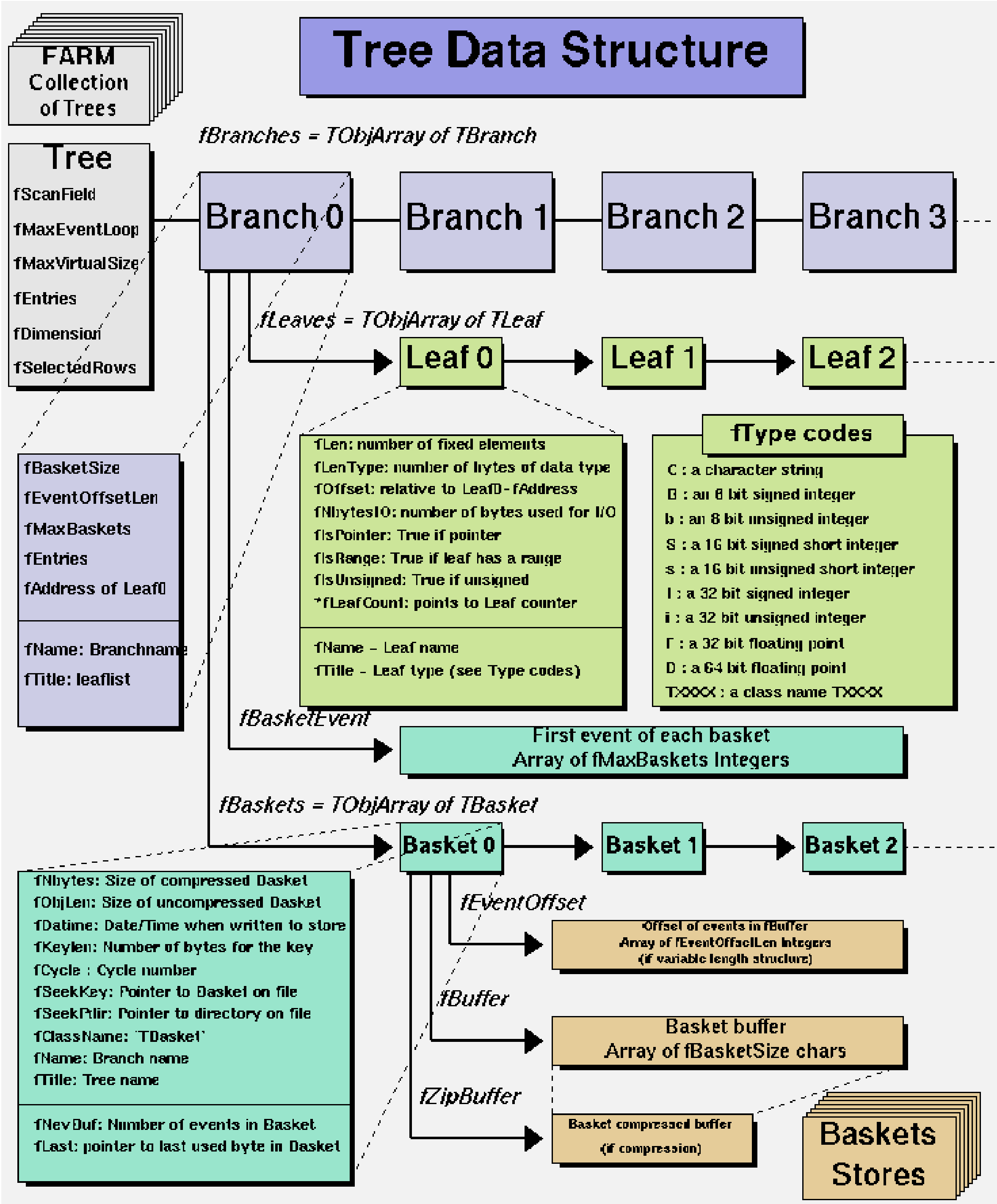


An example of a **ROOT** based event structure is described in NA49 classes. The logical structure of the classes is shown in this GIF file.

A **Tree** (class `TTree`) is made of branches. Each branch (class `TBranch`) is described by its leaves (class `TLeaf`). The leaves can be simple variables, structures, arrays or objects. An array may be of variable length, the length itself being a variable in the same branch or another branch. Branches will in general be objects. However, we thought important to also support variables and structures for these applications not yet converted to C++ and objects. A structure, for example, could be a simple C structure or a list of variables in a Fortran common block.

When the fruits of one branch (detector data) are ready to be picked, they are collected into baskets (class `TBasket`). When the **Baskets** become full, they go to the store, the file.

Each **Branch** will go to a different buffer (**Basket**). The TPC buffers will be written maybe after every event, whereas Trigger info buffers may be written only after every few hundred events and event flag buffers may be kept in memory. The different buffers can be organized to be written to the same file or to different files.



It is easy and efficient for an interactive analysis program to make selections on the event flag and then only read buffers of the selected events. This mechanism is also well suited for parallel architectures. Note that this scheme allows also insertion of a new **Branch** at any time in an existing file or set of files.

When queries are executed on one or more variables or objects, only the branch buffers containing these variables are read into memory.

Data are in general processed on different architectures with different memory sizes. In case the analysis is performed on a parallel architecture with a lot of memory, as many buffers as possible are kept in memory, maybe even all buffers.

The **Tree** data structure allows direct access to any event, direct access to any branch or any leaf even in the case of variable length structures.

How to Write Objects to a File

Objects derived directly or indirectly from the class `TObject` can be made persistent (i.e. written to a file). Before writing an object to a file, you must first create a new file or open an existing file in `UPDATE` or `RECREATE` mode. For example,

- TFile f1("file1.root","NEW");
- TFile f2("file2.root","RECREATE");
- TFile f3("file3.root","UPDATE");

Let's assume *obj* inherits from `TObject`. Objects in a file are identified by a key (see class `TKey`). A key is itself an object with all the necessary information to locate an user object in a file (its name, a title, size, position and a few other parameters). A file has a directory consisting of the list of keys. A key is automatically created by an operation such as *obj*->**`Write`**("*keyname*"). If the file compression attribute is set (`TFile::SetCompressionLevel`), the buffer holding an object is compressed before being written to the file. **`Write`** is a member function of the `TObject` class (`TObject::Write`) and perform the following operations:

- It creates a buffer object *b* (see class `TBuffer`).
- It fills the buffer by invoking *obj*->**`Streamer(`***b***`)`** (see below).
- It writes the buffer to the file.
- It adds a new key with the name "*keyname*" to the list of keys.

The `Streamer` function is responsible for scanning the object data structure and copy the data to the buffer. This function may be user-written, but is, in general, automatically generated by the **rootcint** dictionary generator. To illustrate how `Streamer` is working, we will take the case of the ROOT class `TShape`. A `TShape` object describes a detector geometry basic element. `TShape` derives from `TNamed` that derives from `TObject`. It also derives from two ROOT attribute classes `TAttLine` and `TAttFill`. The `TShape` object has a few data members that are basic types and also a pointer to a material object. `Streamer` includes the code to read and write an object. We concentrate for the time being only on the writing part.

```
//______________________________________________________________________
void TShape::Streamer(TBuffer &b)
{
   // Stream a TShape object
   if (b.IsReading()) {
     Version_t v = b.ReadVersion();
     TNamed::Streamer(b);
     TAttLine::Streamer(b);
     TAttFill::Streamer(b);
     b >> fNumber;
     b >> fVisibility;
     b >> fMaterial;
  } else {
     b.WriteVersion(TShape::IsA());
     TNamed::Streamer(b);
     TAttLine::Streamer(b);
     TAttFill::Streamer(b);
     b << fNumber;
     b << fVisibility;
     b << fMaterial;
  }
}
```

A statement like **b << fNumber** means: encode data member with name **fNumber** into buffer **b**. The `TBuffer` class defines the operators << and >> for all basic data types, but also for pointers to objects. If *fList* is a pointer to a list of objects, then the `Streamer` function for all the objects in the list is automatically called. The code corresponding to **`b <<`** *fObj* is inlined for efficiency.

Let's now see what **b << fMaterial** is doing. `fMaterial` is a pointer to the material object. However, many shapes may reference the same material and we may want to write in the same buffer the complete list of all shapes, still writing only one copy of the referenced material. This is done automatically. `TBuffer` keeps a table of pointers to the objects already written in this buffer. When a reference to an already saved object is encountered, only the serial number of the object in the table is written, not the object itself.

The statement `TNamed::Streamer` invokes the `Streamer` function of the `TNamed` class responsible for saving its own data members. **`TNamed`**, in turn, will call `TObject::Streamer`. Note that the first operation is to write the class version identifier of the `TShape` class. This consists of two bytes and corresponds to the Class version ID given in the **ClassDef** statement in the `TShape` include file. This version ID can be used in the reading part to take into account possible (and likely) changes in the class definition. Therefore ROOT has an overhead of two bytes per level of inheritance for each object. This is necessary to support a complete and safe schema evolution. A mechanism with one single identifier per object would not be sufficient.

Most of the time, the `Streamer` function generated by **rootcint** is appropriate. You will have to implement your own function in the following cases:

• One data member is a pointer to something not-derived from `TObject`. For example, in case of a pointer to an array of **floats**, `TBuffer` has no way to know the length of the array. Look into the `TBuffer` class to see the list of member functions for writing arrays.

• One data member is a C structure. It is up to you to dismantle the structure in as many statements like **b << struct.fX**. We recommend anyhow to change your C structures to C++ classes.

• One data member is a multi-dimensional array (eg. `fMatrix[3][4]`). To write/read such arrays, use `TBuffer:WriteFastArray` and `TBuffer:ReadFastArray` respectively.

• You do not want to write all class data members. Note however, that if **rootcint** processes your class, the generated function will not save data members for which the first character in the associated comment field is "!", e.g.: `fCacheVar; //! cache variable.`

• You want to write additional information.

When your object derives from `TNamed`, you may omit the parameter *keyname*. For example, for a `TShape` object, one could write: *obj*->`Write()`. The key created in this case will get the name from the `TNamed` object. If you write a key with a name already existing in the file, a new cycle (a la VMS) is created. You can use `TFile::ls` or `TFile::Map` to see the list of all records in a file.

How to Read Objects from a File

There are two ways to read an object from a file.

• Via TFile::Get or TDirectory::Get if you are in a sub-directory. Assume an existing file **f** (created by **TFile f("myfile.root")**). You can list all the keys (persistent objects in the file) via **f.ls()** or via the browser (see TBrowser). To read an object in memory (example a TShape object), you can do:

```
TShape *shape = (TShape*)f.Get(shape_name);
```

f.Get will create an object of class TShape with the following sequences of operations:

• Find the key (eg. shape_name) in the list of keys.

• Create a TBuffer object.

• Read the buffer from the file.

• Create an empty object by calling the default constructor for the referenced class in TKey.

• Call the **Streamer** function for this new object.

In case of an object with multiple cycles, one can return a cycle (a la VMS) with **f.Get(name;cycle)**.

• Via TKey::Read. The first method via TDirectory::Get is interesting to return an object by its name in complete random access. In case of a very long list of objects to be processed sequentially (this could be a list of events), it is simpler and more efficient to scan the list of keys directly. The example below illustrates at the same time the use of an **iterator** to loop on all keys of a file.

```
TIter nextkey(f.GetListOfKeys());
TKey *key;
while (key = (TKey*)nextkey()) {
   Event *event = (Event*)key->Read();
   event->Process();
{
```

What happens if your job crashes while you write a file?

In case a program does not terminate in a clean way, the file directory is may be not written at the end of the file. Next time you use the file, ROOT will automatically detect this situation and will recover the directory by scanning sequentially the list of keys in the file. If the file has been opened in UPDATE mode, the recovered directory will be automatically written to the file. This automatic recovery procedure is possible because of redundant information written to the file. In case you write large **TTrees**, you may have large buffers in memory. In case of a job crash, you may loose a lot of data. A mechanism has been foreseen for this case. See TTree::AutoSave.

How to Draw objects

You draw an object via **object->Draw(option)**. This function adds a reference to the object in the list of objects to be drawn in the current pad or canvas. If a canvas does not exist, a default canvas (named **c1**) is automatically created.

A canvas object (see TCanvas) has an associated screen window (unless you are in **batch** mode) and a list of pads (see TPad). The canvas itself is a pad. You do not need to create a pad in case you want to get only one pad. Pads are resizable and moveable areas in the canvas. Pads in turn may contain other pads to build a hierarchy. Pads may be automatically generated by TPad::Divide. In case a canvas or pad is resized, its contents are automatically redrawn.

An instruction like **object->Draw()** in fact does not draw directly. It simply adds a reference to **object** in the list of pad objects. For each object drawn, a **TObjLink** object is created. This **TObjLink** object contains a reference to **object**. This means that if the content of **object** changes, your picture will change. When the canvas/pad must be painted (following a moose-driven resize operation or by calling TCanvas::Update), the TPad::Paint function is called. This function, in turn, scans the list of all objects and for each object invokes **object->Paint(option)**. The parameter "option" corresponds to the parameter given in the **object->Draw(option)** call.

The TCanvas::Update function is automatically called in a ROOT interactive session when **carriage return** is typed. When executing a macro, one can force a screen update by calling **gPad->Update()**, where **gPad** is a global variable pointing to the current pad. You can change the current pad with **pad->cd()** where **pad** is the pointer to an existing pad.

The **object->Paint()** function may invoke other utility functions from the TPad class to paint basic primitives (see for example, TPad::PaintLine, TPad::PaintPolyLine, TPad::PaintText, etc..), or create higher level objects and invoke their **Paint** member function (see for example TH1::Paint).

When an object has been drawn, one can redefine its drawing option by invoking

```
object->SetDrawOption(newoption).
```

When a pad is created, the user coordinates are by default set to the range [0,1] for both x and y. You can use TPad::Range to specify different values. When drawing high level objects such as histograms or graphs, the pad range is automatically set by the **Paint** member function of these classes.

See the list of utility functions in TPad to set or get parameters. You can set a **logarithmic** scale in x and/or y and/or z via TPad::SetLogx, etc.

Use functions such as TPad::XtoPixel or TPad::PixeltoX to convert to pixel coordinates from user coordinates and vice-versa.

How to Pick objects

When the moose is moved or a button pressed/released, the TCanvas::HandleInput function scans the list of objects in all its pads and for each object invokes **object->DistancetoPrimitive(px, py)**. This function computes a "distance" to an object from the moose position at the pixel position px,py and return this distance in pixel units. The selected object will be the one with the shortest computed distance. To see how this work, select the **Event Status** item in the canvas **Options** menu. ROOT will display one status line showing the picked object. If the picked object is, for example an histogram, the status line indicates the name of the histogram, the position x,y in histogram coordinates, the channel number and the channel content.

If you click on the left moose button, the **object->ExecuteEvent(event, px, py)** function is called.

If you click with the right moose button, a context menu (see TContextMenu) with the list of possible actions for this object is shown. This context menu is automatically built by ROOT from the class dictionary by selecting the member functions having the string ****MENU**** in the comment field. Remember that the class dictionary is built by parsing your class header files.

The middle button (or left+right on a 2-buttons moose) can be used to change the current pad to the pointed pad. The current pad is always highlighted. Its frame is drawn with a special color (see TCanvas:SetHighLightColor).

Various examples of **DistancetoPrimitive** or **ExecuteEvent** functions can be seen in the ROOT source code. To users who want to write such functions, we strongly recommend to look at existing implementations in the ROOT source code. In particular, have a look at TLine::ExecuteEvent and TBox::ExecuteEvent. Note that these functions may, in turn, invoke other objects such functions in case an object is drawn using other objects. You can also exploit at best the virtues of inheritance. See for example, how the class TArrow (derived from Tline) or TPaveText (derived from TPave, itself derived from TBox) use or redefine the picking functions in their base classes.

For example in the case of an event display program, the **Event::DistancetoPrimitive** may loop on the list of detectors and tracks and for each detector/track object invokes its **DistancetoPrimitive**. See, for example TGenerator::DistancetoPrimitive.

The moose position is always given in pixel units in these two functions. px=0 and py=0 correspond to the top-left corner of the canvas. Here we have followed the standard convention in windowing systems. Note that user coordinates in a pad have the origin at the bottom-left part of the pad. One can use transformation functions such as TPad::PixeltoX, TPad::XtoPixel,etc. to convert to/from pad coordinates.

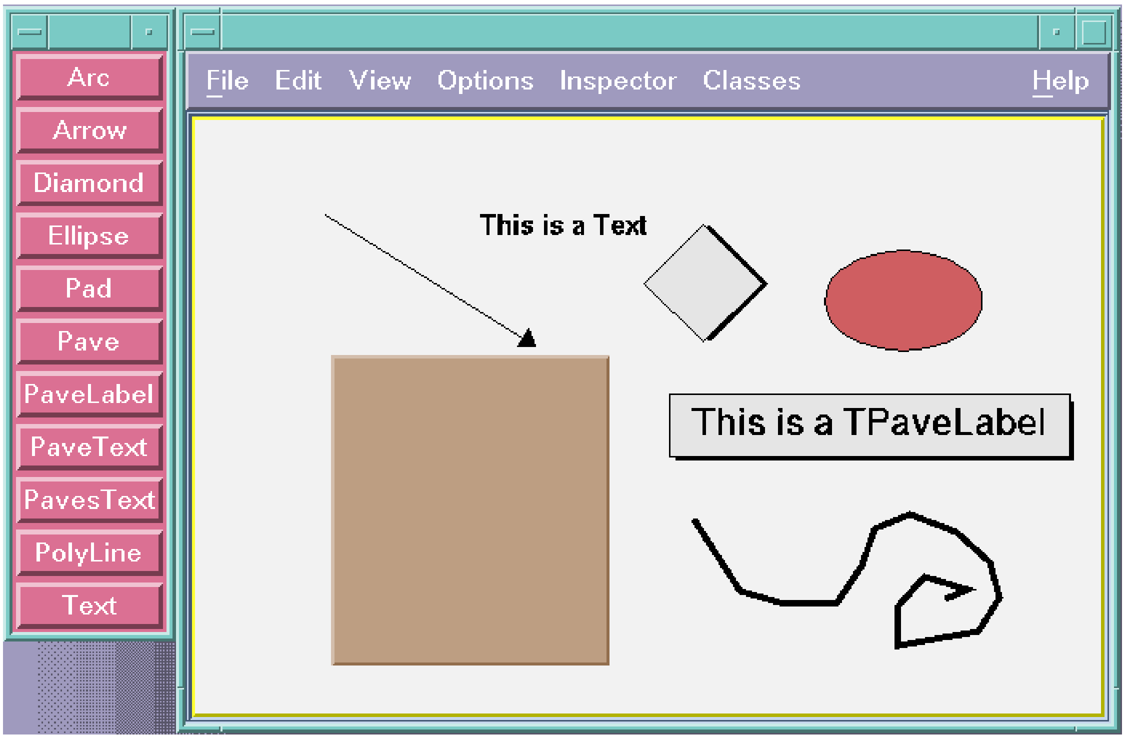
If a named object of a class **MyClass**(derived from) is inserted in a pad, you can get a pointer to this object with:

```
MyClass *obj = (Myclass*)pad->GetPrimitive(objectname);
```

How to Use the Graphics Editor

ROOT has a simple built-in graphics editor to draw and edit basic primitives starting from an empty canvas or on top of a picture (eg. histogram).

The editor is started by selecting the **Editor** item in the canvas **Edit** menu. A menu appears into an independent window (see in the picture below).



You can create the following graphics objects:

• An **arc** of circle. Click on the center of the arc, then move the moose. A rubberband circle is shown. Click again with the left button to freeze the arc.

• An **arrow**. Click with the left button at the point where you want to start the arrow, then move the moose and click again with the left button to freeze the arrow.

• A **Diamond**. Click with the left button and freeze again with the left button. The editor draws a rubber band box to suggest the outline of the diamond.

• An **Ellipse**. Proceed like for an **arc**. You can grow/shrink the ellipse by pointing to the sensitive points. They are highlighted. You can move the ellipse by clicking on the ellipse, but not on the sensitive points. If, with the ellipse context menu, you have selected a fill area color, you can move a filled-ellipse by pointing inside the ellipse and dragging it to its new position. Using the contextmenu, you can build an arc of ellipse and tilt the ellipse.

• A **Pad**. Click with the left button and freeze again with the left button. The editor draws a rubber band box to suggest the outline of the pad.

• A **PaveLabel**. Proceed like for a pad. The type in the label to be put in the box. Then type carriage return. The text will be redrawn to fill the box.

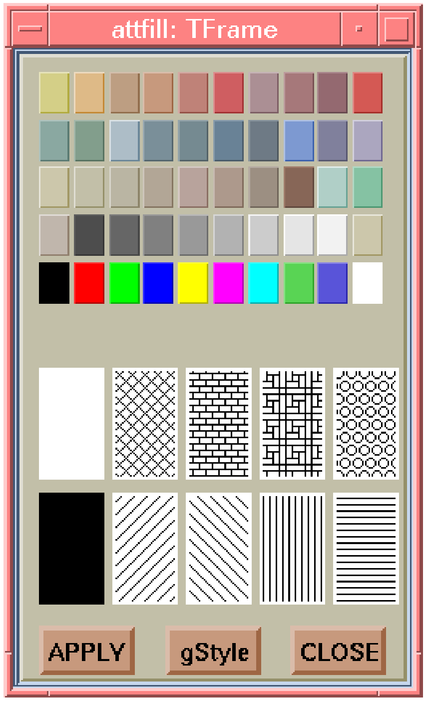
• A **PaveText** or **PavesText**. Proceed like for a pad. You can then click on the PaveText object with the right moose button and select the option **AddText**.

• A **PolyLine**. Click with the left button for the first point, move the moose, click again with the left button for a new point. Close the polyline with a double-click. To edit one vertex point, pick it with the left button and drag to the new point position.

• A **Text string**. Click with the left button where you want to draw the text, then type in the text terminated by carriage return. To move the text, point on it keeping the left moose button pressed and drag the text to its new position. You can grow/shrink the text if you position the moose to the first top-third part of the string, then move the moose up or down to grow or shrink the text respectively. If you position near the bottom-end of the text, you can rotate it.

You can edit an existing object in a canvas by clicking with the right moose button on this object. A pop-up menu is shown with a list of possible actions on this object. You will notice that most graphics objects derive from one or several attribute classes TAttLine, TAttFill, TAttText or TAttMarker.

You can edit these attributes by selecting the corresponding item in the pop-up menu. For example selecting **SetFillAttributes** displays a panel with the color palette and fill area types. The name and class of the object being edited is shown in the bar title of the panel.



Once you are happy with your picture, you can select the **Save as canvas.C** item in the canvas **File** menu. This will automatically **generate a macro** with the C++ statements corresponding to the picture. This facility also works if you have other objects not drawn with the graphics editor (histograms for example). You can see the macro automatically generated from the canvas above.

How to use the Histogram classes

The histogram classes are named **THdp** where **d**(dimension) can be 1, 2 or 3. The precision **p** can be C (for Char), S (for Short), F (for Float) or D (for Double). For all dimensions, both **fix and variable bin size** options are supported.

For example, TH1F is a class for 1-D histograms where a Float is used to increment the sum of weights for each channel. That means that you can create histograms with 1, 2, 4 or 8 bytes per channel.

TH2D is a 2-D histogram with a Double word to store the sum of weights. In addition, a special class TProfile handles profile histograms. All these histogram classes derive from the base class TH1. Therefore, most functions described below can be found in the documentation of TH1. The TH1 class, in turn, derives from TNamed. Histograms are identified by a name and they have a title. TH1 also derives from the standard ROOT attribute classes:

- TattLine: describing the line attributes.
- TAttFill: describing the fill area attributes.
- TAttmarker: describing the marker attributes.

The TProfile class derives from TH1D.

The **TH2p** classes derive from the corresponding **TH1p** classes ans also from the auxiliary class TH2.

The **TH3p** classes derive from the corresponding **TH1p** classes ans also from the auxiliary class TH3.

The TH1C class can be used to create histograms where a maximum sum of weights (or number of entries per bin in case weights are equal to 1) is less than 256.

The TH1S class has a similar limit at 65536. TH1F is the most frequent histogram class (equivalent to **HBOOK1**). It uses a 32 bits floating point to store the sum of weights per channel.

The TH1D class may be used where the sum of weigths requires high precision. This may be the case when the dynamic of weights is large.

Creating histograms

An histogram is created by invoking one of the class constructors. For example, a 1-D histogram with a fix number of bins is created with:

```
TH1F *hfix = new TH1F("hfix","hfix title",nbins,xlow,xup);
```

and a variable bin size histogram is created with:

```
TH1F *hvar = new TH1F("hvar","hvar title",nbins,xbins)
```

where **xbins** is an array of **nbins+1** floats containing the low-edge of the **nbins** first bins and the upper-edge of the last bin.

2-D histograms are created in the same way (4 different constructors for each precision) with fixed/variable bins along x and/or y. When an histogram is created, it is added in the list of objects in memory in the current directory (see Writing histograms to a file below).

Filling histograms

Histogram of all types are filled via the **hist->Fill(..)** functions:

- TH1::Fill(x) increment by 1 the bin corresponding to x.
- TH1::Fill(x,w) increment by w the bin corresponding to x.
- TH1::Fill(x,y) increment by 1 the cell corresponding to x and y.
- TH1::Fill(x,y,w) increment by w the cell corresponding to x and y.
- TH1::Fill(x,y,z), etc.. for 3-D hsitogarms.

In all cases, parameters x and y are of type Float_t (32 bits floating point). the parameter w is of type Stat_t (64 bits floating point).

In case, the function TH1::Sumw2 has been called before filling the histogram, the sum of squares of weights is also incremented for each channel or cell. At the difference of **HBOOK/PAW**, the **Fill** function always increments the histogram data to later compute correctly the **mean value** and the **standard deviation**.

Drawing histograms

An histogram of any type is drawn by TH1::Draw(option). Many different options are supported and are described in this function. Once an histogram has been drawn, one can modify its attributes or drawing options by using the contextmenu item **DrawPanel**.

Writing histograms to a file

Like any other ROOT objects, histograms can be written to a file. Example: **hist->Write()**. When an histogram is created, a reference to this histogram is added in the list of objects of the current directory in memory. If no file is opened at the time of the creation, the histogram is added to the default list of objects in memory. If a file exists, the histogram is added to the list of memory objects associated to this file. To write the full lists of histograms stored in memory for this file, you do:

```
file->Write()
```

where **file** is the pointer to the file object.

Reading histograms from a file

Assume you have opened a file with, for example:

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

you can **f.ls()** to see the list of objects in this file. In an interactive session, you can read directly an histogram in memory, just by typing its name, or **name->Draw()**. ROOT will automatically create a pointer to this object in memory. The name of the pointer is the name of the object in the file. Note that this is a ROOT extension to C++.

The recommended way (working in standard C++ and interactively) is to do:

```
TH1F *hname = (TH1F*)f.Get(hname)
```

In this case, it is your responsability to make the correct cast operation. You can also use the browser (see TBrowser). Clicking on a histogram, forces the histogram in memory and draws it. An histogram is read in memory in the list of memory objects associated to the current directory. This allows for histograms with the same name in different files.

Fitting histograms

Use TH1::Fit to fit histograms. Fitting in ROOT is based on the TMinuit class. By default, when you invoke TH1::Fit, the fitted function object replaces the previously fitted function in the list of fitted objects for this histogram. You can have several associated fitted functions if you specify the fit option "+".

By default, the specified function is fitted in the full histogram range. If you have set a range for the fitted function (see TF1::Range), you can take this range into account by specifying the option "R".

You can also select the histogram contextmenu item **FitPanel** for a visual inspection of the fit options and to set the fit range via a **slider**.

Look at the tutorials fit1 and multifit to see examples.

Other histogram facilities

A long list of histogram service functions is provided. See class TH1 for a complete list.

How to Fit Histograms or Data Points

Fitting in ROOT is based on the TMinuit class. See in particular Basic Concepts of Minuit. The user has the choice of using directly the **TMinuit** class, as illustrated in the last section below or use specialized functions provided in the histogram class TH1 or the graph classes TGraph and TGraphErrors.

The **Minuit** package acts on a multiparameter Fortran function **FCN**. In the ROOT implementation, the function **FCN** is defined via the TMinuit::SetFCN member function when an Histogram.Fit command is invoked. The value of **FCN** will in general depend on one or more variable parameters.

To take a simple example, in case of ROOT histograms (classes TH1C,TH1S,TH1F,TH1D) the Fit function defines the Minuit fitting function as being H1FitChisquare. H1FitChisquare calculates the chisquare between the user fitting function (gaussian, polynomial, user defined,etc) and the data for given values of the parameters. It is the task of MINUIT to find those values of the parameters which give the lowest value of chisquare.

Fitting 1-Dim histograms

Fitting histograms is done via TH1::Fit. The name of the fitted function (the model) is passed as first parameter. This name may be one of the ROOT pre-defined function names or a user-defined function.:

With pre-defined functions

The following functions are automatically created by ROOT when any fitting function is invoked (eg, TH1::FIT):

- **gaus**. A gaussian with 3 parameters: f(x) = **p0***exp(-0.5*((x-**p1**)/**p2**)^2)).
- **expo**. An exponential with 2 parameters: f(x) = exp(**p0**+**p1***x).
- **polN**. a polynomial of degree N: f(x) = **p0** + **p1***x + **p2***x^2 +...

As an example, the following command will fit histogram object hist with a gaussian. By default, the fitting function object will be added to the histogram structure and it will be drawn in the current pad. For pre-defined functions, there is no need to set initial values for the parameters. ROOT will do it automatically for you.

Root > hist.Fit("gaus");

With user-defined functions

You can create a TF1 object. The **TF1** function may be created out of existing known expressions (see TFormula). For example, the following command creates a function called "myfit" with 3 parameters in the range between 0 and 2.

Root > TF1 *myfit = new TF1("myfit","[0]*sin(x) + [1]*exp(-[2]*x)", 0, 2);

Now, we can set parameter names (optional) and parameter start values (mandatory).

```
Root > myfit->SetParName(0,"c0");
Root > myfit->SetParName(1,"c1");
Root > myfit->SetParName(2,"slope");
Root > myfit->SetParameter(0, 1);
Root > myfit->SetParameter(1, 0.05);
```

Root > myfit->SetParameter(2, 0.2);

We are now ready to fit:

Root > hist->Fit("myfit");

You can also create your own fitting function. This function must have 2 parameters:

38. **Double_t *x**: a pointer to the variables array. This array must be a 1-dim array in case of a 1-dim histogram, a 2-dim array for a 2-dim histogram, etc.
39. **Double_t *par**: a pointer to the parameters array. par will contain the current values of parameters when it is called by the **FCN** function.

The following macro **fitexample.C** illustrates how fit a 1-dim histogram stored in a ROOT file with a user-defined function.

```
//_____________________macro fitexample.C__________________________
```

```
Double_t fitf(Double_t *x, Double_t *par)
{
   Double_t arg = 0;
   if (par[2]) arg = (x[0] - par[1])/par[2];
   Double_t fitval = par[0]*TMath::Exp(-0.5*arg*arg);
   return fitval;
}
void fitexample()
{
   TFile *f = new TFile("hsimple.root");

   TH1F *hpx = (TH1F*)f->Get("hpx");
   TF1 *func = new TF1("fit",fitf,-3,3,3);
   func->SetParameters(500,hpx->GetMean(),hpx->GetRMS());
   func->SetParNames("Constant","Mean_value","Sigma");
   hpx->Fit("fit");
}
```

Fitting a sub-range of the histogram bins

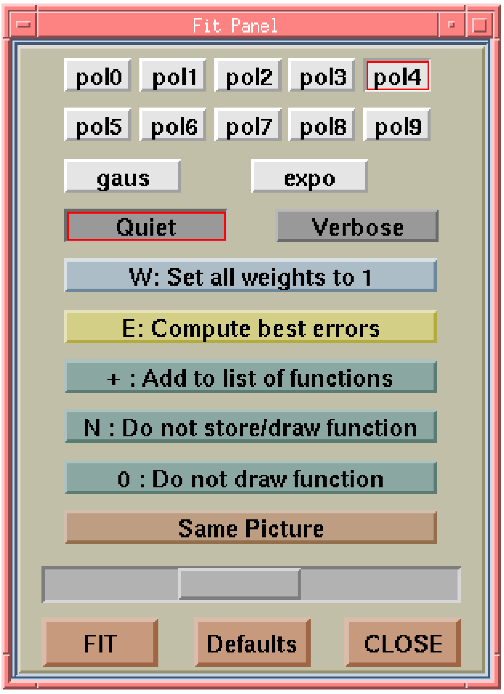
By default,TH1::Fit will fit the function on the defined histogram range. You can specify the option "**r**" in the second parameter to restrict the fit to the range specified in the TF1object. For more complete examples, see A simple fitting example and Fitting histogram sub-ranges.

Fitting several functions on the same histogram

The example Fitting histogram sub-ranges also illustrates how to fit several functions to the same histogram. By default, a Fit operation deletes the previously fitted function in the histogram data structure. You can specify the option "+" in the second parameter of TH1::Fit to add the newly fitted function to the existing list of fitted functions for this histogram. Note that the fitted function(s) are saved with the histogram when the histogram is written to a ROOT file.

Using the histogram FitPanel

To fit an histogram drawn in a pad, you can click on the histogram contour with the right mouse button and select the **FitPanel** option. This function from the TFitPanel class will display the following menu:



Fitting non-equidistant data points

To fit non-equidistant data points, you can use the services of the classes TGraph or TGraphErrors. Examples of creation of such objects are given in:

• A simple Graph example.

• Example of a graph with error bars.

To fit such objects, use TGraph::Fit. For example, to fit the graph called "**gr** in the above examples, with a polynomial of degree 4, you do:

Root > gr->Fit("pol4");

Like for histograms above, you can use the interactive **Fitpanel** option in the graph context menu.

Fitting in general using the class TMinuit

The following example (part of the tutorials: Using Minuit to fit non-equidistant points)) macro illustrates how to fit in general by using the services of the TMinuit class only.

```
//
//   Example of a program to fit non-equidistant data points
//   =======================================================
//
//   The fitting function fcn is a simple chisquare function
//   The data consists of 5 data points (arrays x,y,z) + the errors in errorsz
//   More details on the various functions or parameters for these functions
//   can be obtained in an interactive ROOT session with:
//    Root > TMinuit *minuit = new TMinuit(10);
//    Root > minuit->mnhelp("*",0)  to see the list of possible keywords
//    Root > minuit->mnhelp("SET",0) explains most parameters
//
Float_t z[5],x[5],y[5],errorz[5];
//______________________________________________________________________________
void fcn(Int_t &npar, Double_t *gin, Double_t &f, Double_t *par, Int_t iflag)
{
  const Int_t nbins = 5;
  Int_t i;
//calculate chisquare
   Double_t chisq = 0;
  Double_t delta;
  for (i=0;i<nbins; i++) {
    delta  = (z[i]-func(x[i],y[i],par))/errorz[i];
    chisq += delta*delta;
  }
  f = chisq;
}
//______________________________________________________________________________
Double_t func(float x,float y,Double_t *par)
{
Double_t value=( (par[0]*par[0])/(x*x)-1)/ ( par[1]+par[2]*y-par[3]*y*y);
return value;
}
//______________________________________________________________________________
void Ifit()
{
// The z values
        z[0]=1;
       z[1]=0.96;
       z[2]=0.89;
       z[3]=0.85;
       z[4]=0.78;
// The errors on z values
        Float_t error = 0.01;
       errorz[0]=error;
       errorz[1]=error;
       errorz[2]=error;
       errorz[3]=error;
       errorz[4]=error;
// the x values
        x[0]=1.5751;
       x[1]=1.5825;
       x[2]=1.6069;
       x[3]=1.6339;
       x[4]=1.6706;
// the y values
        y[0]=1.0642;
       y[1]=0.97685;
       y[2]=1.13168;
       y[3]=1.128654;
       y[4]=1.44016;
  TMinuit *gMinuit = new TMinuit(5);  //initialize TMinuit with a maximum of 5 params
  gMinuit->SetFCN(fcn);
  Double_t arglist[10];
  Int_t ierflg = 0;
  arglist[0] = 1;
  gMinuit->mnexcm("SET ERR", arglist ,1,ierflg);
// Set starting values and step sizes for parameters
   static Double_t vstart[4] = {3, 1 , 0.1 , 0.01};
  static Double_t step[4] = {0.1 , 0.1 , 0.01 , 0.001};
```

```
   gMinuit->mnparm(0, "a1", vstart[0], step[0], 0,0,ierflg);
   gMinuit->mnparm(1, "a2", vstart[1], step[1], 0,0,ierflg);
   gMinuit->mnparm(2, "a3", vstart[2], step[2], 0,0,ierflg);
   gMinuit->mnparm(3, "a4", vstart[3], step[3], 0,0,ierflg);
// Now ready for minimization step
    arglist[0] = 500;
   arglist[1] = 1.;
   gMinuit->mnexcm("MIGRAD", arglist ,2,ierflg);
// Print results
    Double_t amin,edm,errdef;
   Int_t nvpar,nparx,icstat;
   gMinuit->mnstat(amin,edm,errdef,nvpar,nparx,icstat);
   gMinuit->mnprin(3,amin);
}
```

Access to the Fit parameters and results

Once an histogram has been fitted, you can access the fitted function parameters via the "Get" functions of the TF1 and TFormula classes. Examples:

```
Root > TF1 *fit = hist->GetFunction(function_name);
Root > Double_t chi2 = fit->GetChisquare();
Root > Double_t p1 = fit->GetParameter(1);
```

Root > Double_t e1 = fit->GetParError(1);

Note that above **p1** refers to the second parameter.

You can enable the display of the fit parameters in the current canvas by selecting the corresponding item in the **Options** canvas tool bar menu. You can also set the display of the fit parameters by default in a ROOT session via:

Root > gStyle->SetOptFit();

Fitting with one or more parameters between some bounds

To set bounds for one parameter, use TF1::SetParLimits:

Root > func->SetParLimits(0, -1, 1);

where **func** is the pointer to the function to be fitted. If you only have the function name, you can get the pointer to this function with:

Root > gROOT->GetFunction(func_name);

How to Create and Fill a Tree

The concept of **Tree** has been introduced at Input/Output with Trees. An example of **Tree** is given in the documentation of the class TTree. A more complete example is given in Event and its I/O performance in ROOT Input/Output Benchmarks.

In this chapter, we will use variants of the program **Event** to illustrate the advantages of **Trees** compared to **the object serialization method** described in How to Write Objects.

It is very important to note that alternating between the methods described below implies NO changes in the object model.

Using the serialization method and creating a key

The serialization method is perfectly appropriate when it does not make sense to access sub-objects components. For example, you write an histogram object and you read it as an histogram. It is unlikely that you want to read back only one part of an histogram.

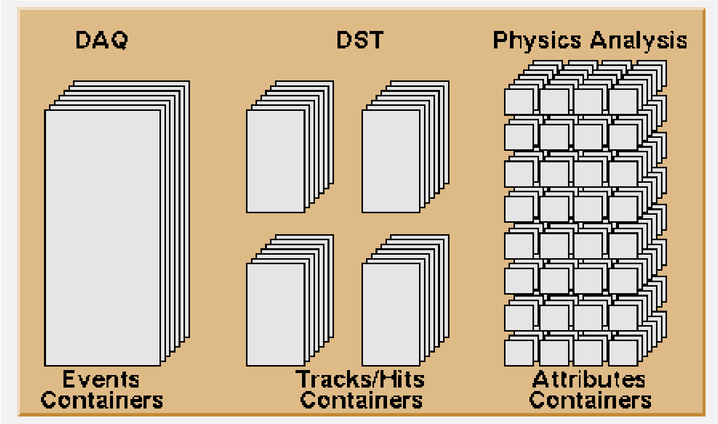
This case does not use a **Tree** at all. Once the event is filled, it is serialized (via **event->Write** in one buffer and written on the file using a key (in this case the string "event" followed by the event number).

This technique is, may be, the one to use in the context of a data acquisition system where you want to be as fast as possible in producing the output file. It is likely that the reconstruction program will need access to the totality of the event information. One variant of this scheme could be to create several keys per event, one key for each of the main detector components.

A system with several keys per event may be interesting during the DST(Data Summary Tape) analysis if you really insist in remaining very conventional. (see also picture below where different levels of event granularity are suggested for each application domain).

However, based on experience and results of benchmarking with different configurations, we strongly encourage users to use **Trees** instead.

```
//________________________The Serialization method_________________________
main(int argc, char **argv)
{
  TFile hfile("Event.root","RECREATE");
   // Fill event, header and tracks with some random numbers
  Event  *event;
  char keyname[16];
  for (Int_t ev = 0; ev < 1000; ev++) {
     Float_t sigmat, sigmas;
     gRandom->Rannor(sigmat,sigmas);
     Int_t ntrack   = Int_t(600 +5*sigmat);
     Float_t random = gRandom->Rndm(1);
     event = new Event();    // Create one event
     event->SetHeader(ev, 200, 960312, random);
     event->SetNseg(Int_t(10*ntrack+20*sigmas));
     event->SetNvertex(1);
     event->SetFlag(UInt_t(random+0.5));
     event->SetTemperature(random+20.);
      //  Create and Fill the Track objects
     for (Int_t t = 0; t < ntrack; t++) event->AddTrack(random);
      //  Encode key name (see TKey) using the event serial number
     sprintf(keyname,"event%d",ev);
      //  Serialize event into one single buffer
     event->Write(keyname);
     delete event;
  }
  hfile.Close();
  return 0;
}
//_______________________________________________________________________________
```



Making a Tree with one single branch

The program below is slightly modified compared to the program above. It creates a **Tree** with one single branch (variable **split** is set to 0). The branch is created during the first pass in the loop. The branch address is reset for the other passes because a new **Event** object is created and the branch address is changed to the new event address.

When the program will terminate, only one key will be written to the file, corresponding to the Tree header. The **tree->Fill** statement generates several buffers (baskets) dumped to the file when they are full. The variable **bsize** specifies the initial basket size. If this size is not enough to accomodate at least one event, it will be resized accordingly.

This method offers only slight advantages compared to the first method.

- It generates files slightly smaller than the first case.
- It minimizes the number of keys.
- Looping on the data set is a bit simpler (see next chapter)
- One can use the TTree::Draw command to histogram a member function of the Event class.

```
//_______________________A Tree with one single branch_____________________
main(int argc, char **argv)
{
  TFile hfile("Event.root","RECREATE");
   // Create a ROOT Tree
  TTree *tree = new TTree("T","An example of a ROOT tree");
  Int_t split = 0;
  Int_t bsize = 64000;
  Event  *event = 0;
    // Create one branch. If splitlevel is set, event is a superbranch
    // creating a sub branch for each data member of the Event object.
  tree->Branch("event", "Event", &event, bsize,split);
   // Fill event, header and tracks with some random numbers
  char keyname[16];
  for (Int_t ev = 0; ev < 1000; ev++) {
     Float_t sigmat, sigmas;
     gRandom->Rannor(sigmat,sigmas);
     Int_t ntrack   = Int_t(600 +5*sigmat);
     Float_t random = gRandom->Rndm(1);
     event = new Event();     // Create one event
     event->SetHeader(ev, 200, 960312, random);
     event->SetNseg(Int_t(10*ntrack+20*sigmas));
     event->SetNvertex(1);
     event->SetFlag(UInt_t(random+0.5));
     event->SetTemperature(random+20.);
      //  Create and Fill the Track objects
     for (Int_t t = 0; t < ntrack; t++) event->AddTrack(random);
      //  Serialize single branch of event into one single basket
     tree->Fill();
     delete event;
  }
  hfile.Close();
  return 0;
}
//_____________________________________________________________________________
```

Making a Tree with many branches

If you run the program above and set **split = 1**, then the statement:

```
tree->Branch("event", "Event", &event, bsize,split);
```

will generate several branches, one for each data member of the Event class. The following rules are applied:

• If a data member is a basic type, it becomes one basic branch of class TBranch.

• If a data member is an object, the data members of this object are also split into branches (one level only). This is , for example the case for the data member **fEvtHdr**.

• If a data member is pointer to an object, a special branch of type TBranchObject is created. This is the case in our example for the data member **fH**, a pointer to an histogram. The **fH** branch will be filled by calling the class **Streamer** function to serialize this object into the branch buffer.

• If a data member is a pointer to a TClonesArray object, one super branch of class TBranchClones is created. In our example, this is the case for the data member **fTracks**. This super branch will have a buffer where to store the number of objects in the array. This super branch has the name of the data member. ROOT will also automatically generate additional branches, one branch for each data member of the class referenced by the array (Track in the example).

This method provides the highest possible level of granularity. You continue obviously to have direct access to any single event. In addition, you can now read selectively a subset of all the branches in your analysis program. This is the subject of the next chapter.

This method requires more memory, because one buffer must be created for each branch. In case, you have memory problems, you should decrease the buffer size (parameter **bsize** in the above program.

Remarks about compression when running in split mode

If the file compression is activated (default), ROOT will compress the branch buffers depending on the level of file compression (set by TFile:SetCompressionLevel).

• If level = 1, only branches containing integers will be compressed.

• If level >= 2, all branches will be compressed with compression level-1.

Making branches by hand

If you are not happy with the way the automatic split algorithm works, you can create branches yourself by invoking directly the TTree::Branch function.

Do not forget, if you do so, to set the addresses for every created branch in the filling loop, in case the addresses change.

How to Read a Tree

There are 2 ways of reading a **Tree** and to make selections. Both ways will be illustrated by using the file generated in the previous section How to Create and Fill a Tree.

The simple way via TTree::Draw

Use this function if you want to quickly histogram a few variables with a very simple selection mechanism. The loop on the **Tree** is automatically done by ROOT.

void Draw(Text_t *varexp, Text_t *selection, Option_t *option,Int_t nevents, Int_t firstevent);

where the parameters have the following meaning:

- **varexp** is an expression with the forms **e1**, **e1:e2** or **e1:e2:e3** to generate 1-D, 2-D or 3-D distributions respectively and where **e1**,**e2**,**e3** are expressions with combinations of the original **Tree** variables. Examples:

```
varexp = x     simplest case: draw a 1-Dim distribution of column named x
       = sqrt(x)            : draw distribution of sqrt(x)
       = x*y/z
       = y:sqrt(x) 2-Dim distribution of y versus sqrt(x)
```

- **selection** is an expression with a combination of the **Tree** variables. In a selection all the C++ operators are authorized. Example:

```
selection = "x<y && sqrt(z)>3.2"
```

- **option** is the drawing option. See TH1::Draw for the list of all drawing options. If option contains the string "**goff**", no graphics is generated.

- **nevents** is the number of events to process (default is all).

- **first** is the first event to process (default is 0).

- By default the temporary histogram created is called **htemp**. If **varexp** contains **>>hnew**, the new histogram created is called **hnew** and it is kept in the current directory. Example:

```
tree.Draw("sqrt(x)>>hsqrt","y>0")
```

will draw sqrt(x) and save the histogram as "hsqrt" in the current directory.

By default, the specified histogram is reset. To continue to append data to an existing histogram, use "+" in front of the histogram name.

```
tree.Draw("sqrt(x)>>+hsqrt","y>0")
```

will not reset hsqrt, but will continue filling.

Look at the tutorial macro ntuple1 for a more detailed example.

The kind of expressions you can specify in this function is restricted on purpose (see class TFormula). Our previous experience with the query processor of **PAW** indicates that we better restrict this kind of selection mechanism to trivial cases if we want our users to trust their results.

Note that this method does not require access to the original class code, if you only want to perform the kind of operations described above.

The general way

Again, at the difference of **PAW**, we believe that the ROOT user must be responsible to organize himself the loop on all events (rows) of a **Tree**. In a loop, in general, one wants to generate many histograms, or correlate variables from one **Tree** with variables of another **Tree**, etc. We reproduce below a slightly simplified version of the macro eventb. This macro illustrates:

- How to loop on a tree.
- How to read a sub-branch only (the branch containing the number of tracks).
- When the selection is positive, how to read all the branches.
- How to histogram the result of a member function of the event object.

```
{
```

```
  gROOT->Reset();
//   Connect file generated in $ROOTSYS/test
  TFile f("Event.root");
//   Create an histogram
  TH1F *hnseg = new TH1F("hnseg","Number of segments for selected tracks",5000,0,5000);
//   Start main loop on all events
  TClonesArray *tracks = 0;
  Event *event = 0;   //pointer to event preset to 0
  TBranch *bntrack = T.GetBranch("fNtrack");
  TBranch *branch  = T.GetBranch("event");
  branch->SetAddress(&event);
  Int_t nevent = T.GetEntries();
  Int_t nselected = 0;
  for (Int_t i=0;i<nevent;i++) {
     bntrack->GetEvent(i);
     if (event->GetNtrack() > 587)continue;
     nselected++;
     T.GetEvent(i);                    //read complete accepted event in memory
     hnseg->Fill(event->GetNseg());  //Fill histogram with number of segments
     tracks = event->GetTracks();    //get pointer to the TClonesArray object
     tracks->Clear();                //clear it
  }
//  Delete ClonesArray and histogram objects
  event->Finish();
//  Draw the histogram
  hnseg->Draw();
}
```

This method requires access to the class implementation when one of the branches is an object.

ROOT version 1.0 includes a new function TTree::MakeCode to automatically generate the code for a **skeleton analysis function**.

How to use Chains

A TChain object is a list of Root files containing the same tree. As an example, assume we have 3 Root files called **file1.root**, **file2.root**, **file3.root**. Each file contains one tree called "T". We can create a chain with the following statements:

```
TChain chain("T");
chain.Add("file1.root");
chain.Add("file1.root");
chain.Add("file1.root");
```

The class TChain is derived from the class tree. For example, to generate an histogram corresponding to the attribute "x" in tree "T" by processing sequentially the 3 files of this chain, we can do:

```
chain.Draw("x");
```

The following statements are a slight variation of the example shown in How to Read a Tree. They illustrate how to set the address of the object to be read and how to loop on all events of all files of the chain.

```
{
  gROOT->Reset();
//   Create the chain as above
  TChain chain("T");
  chain.Add("file1.root");
  chain.Add("file1.root");
  chain.Add("file1.root");
//   Create an histogram
  TH1F *hnseg = new TH1F("hnseg","Number of segments for selected tracks",5000,0,5000);
//   Specify address where to read the event object
//   In the program writing the files, the event was stored in a branch called "event"
  Event *event = 0;   //pointer to event preset to 0
  chain.SetBranchAddress("event", &event);
//   Start main loop on all events
//   In case you want to read only a few branches, use TChain::SetBranchStatus
//   to activate/deactivate a branch.
  Int_t nevent = chain.GetEntries();
  for (Int_t i=0;i<nevent;i++) {
     chain.GetEvent(i);              //read complete accepted event in memory
     hnseg->Fill(event->GetNseg());  //Fill histogram with number of segments
  }
//  Draw the histogram
  hnseg->Draw();
}
```

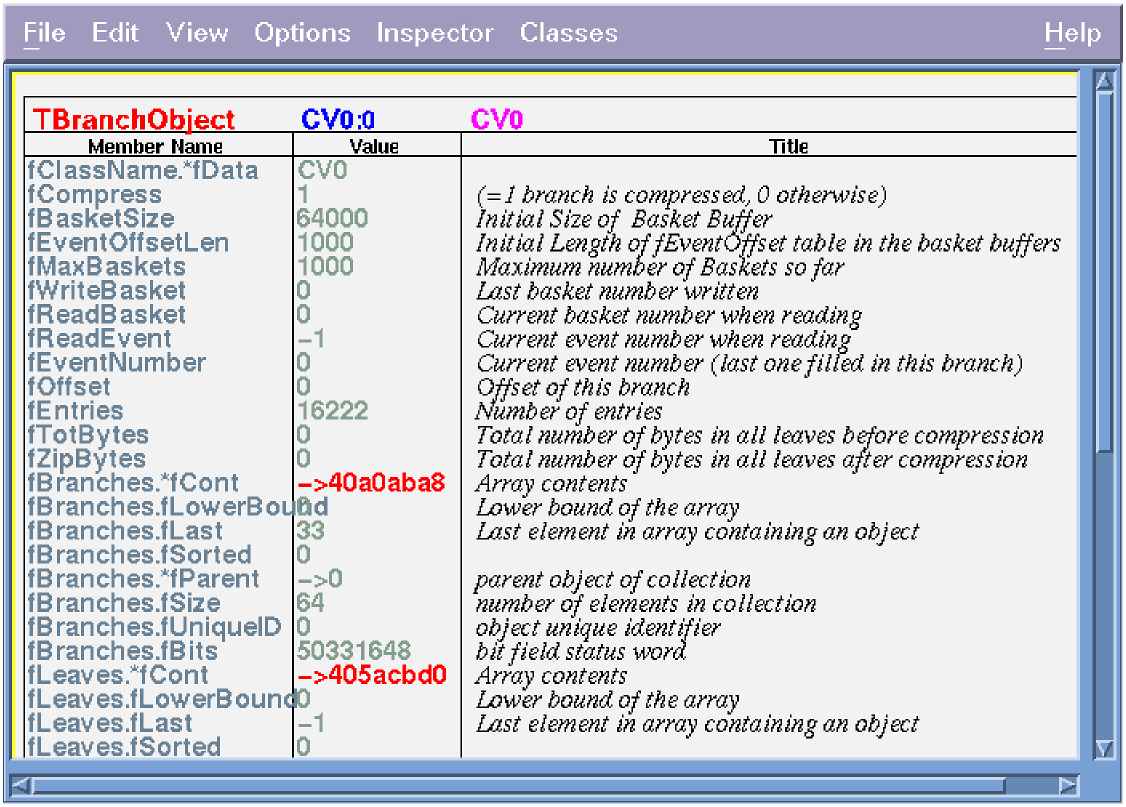
How to Debug and understand a class structure

The class structure and Object-Orientation make possible more modular programming. However, programs consisting of several hundred classes and several hundred thousand lines of code may be difficult to understand, in particular for novice users. ROOT provides a set of tools to help understanding an existing class structure, a data structure in memory or files.

• TClass::Draw can be used to draw the **inheritance tree** for a given object. All ROOT classes are documented with their associated inheritance tree (see for example TPaveText). This tree can be generated interactively via the command: **object->DrawClass**() or by selecting the **DrawClass** item when pointing with the right moose button to an object in a canvas. The tree is automatically generated when a given class is processed through the documentation tool (class THtml). For example:

```
THtml html;
html.MakeClass(myclass);
```

• TObject::Inspect can be used to view the contents of a given object. A table is shown where, for each data member, its name, current value and its title are given.



TBranchObject	CV0:0	CV0
Member Name	**Value**	**Title**
fClassName.*fData	CV0	
fCompress	1	(=1 branch is compressed, 0 otherwise)
fBasketSize	64000	Initial Size of Basket Buffer
fEventOffsetLen	1000	Initial Length of fEventOffset table in the basket buffers
fMaxBaskets	1000	Maximum number of Baskets so far
fWriteBasket	0	Last basket number written
fReadBasket	0	Current basket number when reading
fReadEvent	-1	Current event number when reading
fEventNumber	0	Current event number (last one filled in this branch)
fOffset	0	Offset of this branch
fEntries	16222	Number of entries
fTotBytes	0	Total number of bytes in all leaves before compression
fZipBytes	0	Total number of bytes in all leaves after compression
fBranches.*fCont	->40a0aba8	Array contents
fBranches.fLowerBound	0	Lower bound of the array
fBranches.fLast	33	Last element in array containing an object
fBranches.fSorted	0	
fBranches.*fParent	->0	parent object of collection
fBranches.fSize	64	number of elements in collection
fBranches.fUniqueID	0	object unique identifier
fBranches.fBits	50331648	bit field status word
fLeaves.*fCont	->405acbd0	Array contents
fLeaves.fLowerBound	0	Lower bound of the array
fLeaves.fLast	-1	Last element in array containing an object
fLeaves.fSorted	0	

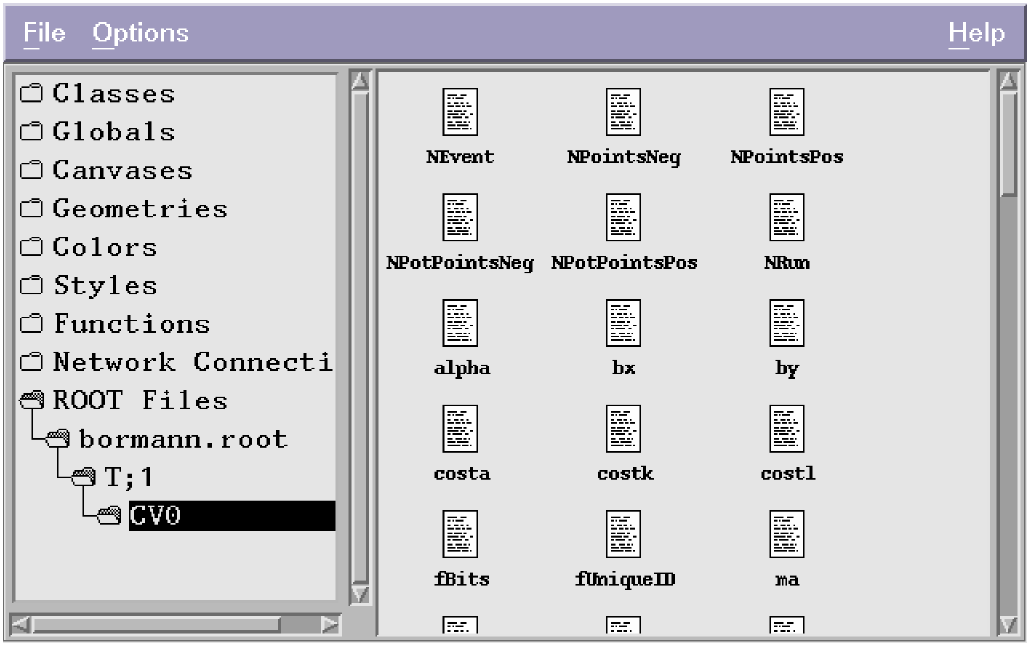
If a data member is a **pointer** to another object, one can click on the pointer and, in turn, inspect the pointed object,etc.

• TObject::Dump is the same as TObject::Inspect, except that the output is on **stdout**. An object dump can be written to a file. Example:

```
     TFile f("hsimple.root"); // connects a file
     f.Dump();  // produces the following dump of the object f
fBEGIN                  64         First used byte in file
fEND                    459500     Last used byte in file
fVersion                9010       File format version
fCompress               1          (=1 file is compressed, 0 otherwise)
fOption.*fData          READ
fUnits                  4          Number of bytes for file pointers
fSeekFree               459414     Location on disk of free segments structure
fNbytesFree             86         Number of bytes for free segments structure
fWritten                0          Number of objects written so far
fSumBuffer              0          Sum of buffer sizes of objects written so far
fSum2Buffer             0          Sum of squares of buffer sizes of objects written so far
*fStream                ->4033f690 Input file stream identifier
```

```
*fOStream              ->0         Output stream (output on same file as fStream)
*fFree                 ->0         Free segments linked list table
fModified              1           true if directory has been modified
fWritable              0           true if directory is writable
fDatimeC.fDatime       139248235   Date (relative to 1995) + time
fDatimeM.fDatime       139248246   Date (relative to 1995) + time
fNbytesKeys            307         Number of bytes for the keys
fNbytesName            120         Number of bytes in TNamed at creation time
fSeekDir               64          Location of directory on file
fSeekParent            0           Location of parent directory on file
fSeekKeys              459107      Location of Keys record on file
*fFile                 ->4033f508  pointer to current file in memory
*fKeys                 ->4033f628  Pointer to keys list in memory
*fMother               ->0         pointer to mother of the list
*fIndex                ->0         Index of the list
*fListHead             ->4033f5f0  Pointer to Objects in this directory
fName.*fData           hsimple.root
fTitle.*fData
fUniqueID              0           object unique identifier
fBits                  50331649    bit field status word
```

• The ROOT general object browser (see TBrowser) can be used to browse collections such as the list of classes, geometries, files and TTrees. A browser can be started from the **Start Browser** item in the canvas **View** menu or by creating a browser object. Example: **TBrowser b**. More than one browser can be active at any time.



• TFile::ls can be used to list objects in a file or directory.

• **list->ls()** lists objects in a collection pointed by **list**.

• **list->Dump()** dumps all objects in a collection pointed by **list**.

• TFile::Map dumps the structure of a file. One line is printed for every record written to the file with the following information:

```
- Date/Time when the object was written.
- Position of the object (starting byte number).
- Number of bytes occupied by the object.
- Object class name.
- Compression factor (if file is compressed).
```

For example, to see the map of the file generated by the tutorial hsimple, you can type the following commands:

```
     TFile f("hsimple.root"); //connect file
     f.Map();  // this produces the following output
970106/120943  At:64        N=150       TFile
970106/120944  At:214       N=28909     TBasket        CX =  1.11
970106/120945  At:29123     N=29046     TBasket        CX =  1.10
970106/120946  At:58169     N=29021     TBasket        CX =  1.10
970106/120946  At:87190     N=28994     TBasket        CX =  1.10
970106/120947  At:116184    N=29069     TBasket        CX =  1.10
```

```
970106/120948  At:145253    N=29101     TBasket        CX =  1.10
970106/120948  At:174354    N=29019     TBasket        CX =  1.10
970106/120949  At:203373    N=28930     TBasket        CX =  1.11
970106/120949  At:232303    N=29040     TBasket        CX =  1.10
970106/120950  At:261343    N=29001     TBasket        CX =  1.10
970106/120951  At:290344    N=29035     TBasket        CX =  1.10
970106/120951  At:319379    N=29003     TBasket        CX =  1.10
970106/120952  At:348382    N=29008     TBasket        CX =  1.10
970106/120953  At:377390    N=29049     TBasket        CX =  1.10
970106/120953  At:406439    N=29025     TBasket        CX =  1.10
970106/120953  At:435464    N=455       TH1F           CX =  1.80
970106/120954  At:435919    N=1724      TH2F           CX =  4.33
970106/120954  At:437643    N=1850      TProfile       CX =  1.55
970106/120954  At:439493    N=19614     TNtuple        CX =  1.74
970106/120954  At:459107    N=307       TFile
970106/121159  At:459414    N=86        TFile
```

• **gObjectTable->Print()**. ROOT keeps tracks of all objects created into a special table in memory. In the development phase of a program, it is frequent to create objects and forget to delete them (**memory leaks**). Printing the object table shows the number of instances for each class currently active in memory. It also shows the total number of bytes occupied by these objects. Example:

```
 gObjectTable->Print(); produces the following output:
Object statistics
class                     cnt    on heap     size    total size    heap size
============================================================================
TH1F                        1          1      448           448          448
TObject                    11         11       12           132          132
TFileType                   6          6       36           216          216
TRint                       1          1      116           116          116
TClassTable                 1          1       12            12           12
TObjectTable                1          1       12            12           12
THashList                   5          5       40           200          200
THashTable                  5          5       36           180          180
TList                      24         24       36           864          864
TStyle                      1          1      328           328          328
TGeometry                   1          1       64            64           64
TGaxis                      1          0      124           124            0
TAxis                       6          3       88           528          264
TUnixSystem                 1          1      256           256          256
TPostScript                 1          1      764           764          764
TOrdCollection              4          4       40           160          160
TEnv                        1          1       24            24           24
TBenchmark                  1          1       52            52           52
TSignalHandler              1          1       20            20           20
TFileHandler                2          2       20            40           40
TCint                       1          1      208           208          208
TFile                       1          1      160           160          160
TROOT                       1          0      244           244            0
TClass                      6          6       64           384          384
TKey                        4          4       80           320          320
TStopwatch                  1          0       56            56            0
TGuiFactory                 2          2       28            56           56
TGXW                        1          1       84            84           84
TRandom                     1          1       32            32           32
TGX11                       1          1      176           176          176
----------------------------------------------------------------------------
Total:                     94         88     3660          6260         5572
============================================================================
```

How to use the PostScript interface

To generate a Postscript (or encapsulated ps) file corresponding to a single image in a canvas, you can:

• Select the **Print PostScript** item in the canvas **File** menu. By default, a Postscript file with the name of the canvas.ps is generated.

• Click in the canvas area, near the edges, with the right mouse button and select the **Print** item. You can select the name of the Postscript file. If the file name is xxx.ps, you will generate a Postscript file named xxx.ps. If the file name is xxx.eps, you generate an encapsulated Postscript file instead.

• In your program (or macro), you can type:

• **c1->Print("xxx.ps")** or **c1->Print("xxx.eps")**

This will generate a file corresponding to the picture in the canvas pointed by **c1**.

pad1->Print("xxx.ps")

prints only the picture in the pad pointed by **pad1**. The size of the Postcript picture, by default, is computed to keep the aspect ratio of the picture on the screen, where the size along x is always 20cm. You can set the size of the PostScript picture before generating the picture with a command such as:

```
TPostScript myps("myfile.ps",-111)
myps.Range(xsize,ysize);
object->Draw();
myps.Close();
```

You can resume writing again in this file with **myps.Open();**. Note that you may have several Postscript files opened simultaneously.

How to Generate the HTML documentation for user classes

All ROOT tutorials as well as the class descriptions have been generated automatically by ROOT itself via the services of the THtml class. Please read this class description and Coding Conventions. The following macro illustrates how to generate the html code for one class using the Make function. This example also shows how to convert to html a macro, including the generation of a "gif" file produced by the macro.

```
{
//
// Create a THtml object. The object can also be accessed via
// the global pointer gHtml
//
THtml html
//
//
// How to generate HTML files for a single class
// (in this example class name is TBRIK),
...
gHtml->MakeClass("TBRIK")
//
// and how to generate html code for all classes, including an index.
//gHtml->MakeAll();
// execute a macro
.x something.C
// Invoke the TSystem class to execute
a shell script.
// Here we call the "xpick" program to capture the graphics window
// into a gif file.
gSystem->Exec("xpick
html/gif/shapes.gif")
// Convert this macro into html
gHtml->Convert("htmlex.C","Automatic
HTML document generation")
```

For more details, in particular, the definition of some **environment variables**, see the documentation of the class THtml.

How to Compile and Link with the ROOT libraries

The ROOT distribution kit includes a directory **root/test** with a suite of test programs to be compiled and linked with the standard ROOT libraries. This directory includes also a **Makefile** specific to each platform to compile and link a given test program. For example, to generate an executable module of the Event test program, you execute the shell script command:

```
make Event
```

This command will:

• Compile Event.C if it has been modified since the previous make.

• Invoke the **rootcint** utility in case the **Event.h** file has been modified and will compile it.

• Generate the shared library **libEvent.so**.

• Generate the executable module **Event**.

To see how this makefile looks on various Unix platforms, look at Makefile for Unix platforms .

If you run an interactive ROOT session, you can **dynamically link** to an existing shared library by executing a command like:

```
gSystem->Load("libEvent.so");
```

In addition, you have interactive access to member functions of the classes that have been processed to **rootcint**. In the same test directory, see examples of macros eventa and eventb. These two macros execute code from the dynamically linked library **libEvent.so**.

How to Convert HBOOK files to ROOT

The ROOT distribution kit includes a program called **h2root** that you can use to convert your HBOOK/PAW histograms or ntuples files into ROOT files. To use this program, you type the shell script command:

```
h2root  hbookfile  rootfile
```

If you do not specify the second parameter, a file name is automatically generated for you. If hbookfile is of the form file.hbook, then the ROOT file will be called file.root.

This program converts HBOOK histograms into ROOT objects of the class TH1F. HBOOK profile histograms are converted into ROOT profile histograms (see class TProfile). HBOOK **row-wise** and **column-wise** ntuples are automatically converted to ROOT **Trees**(see TTree). Some HBOOK column-wise ntuples may not be fully converted in case one of the columns is an array with fix dimensions(eg. var[6]) or is a multi-dimensional array.

HBOOK integer identifiers are converted into ROOT named objects by prefixing the integer identifier with the letter "h" if the identifier is a positive integer and by "h_" if it is a negative integer identifier.

In case of row-wise or column-wise ntuples, each column is converted to a branch of a **Tree**.

Note that **h2root** is able to convert HBOOK files containing several levels of sub-directories.

Once you have converted your file, you can look at it and draw histograms or process ntuples using the ROOT interactive module. An example of session is shown below:

```
TFile f("hbookconverted.root"); // this connects the file hbookconverted.root
h10.Draw(); //display histogram named h10 (was HBOOK id 10)
h30.Draw("var"); //display column "var" from ntuple h30
```

You can also use the ROOT browser (see TBrowser) to inspect this file.

The chapter How to read a Tree explains two ways to read a **Tree**. ROOT version 1.0 will include a new function TTree::MakeCode to automatically generate the code for a **skeleton analysis function**. With ROOT version 0.90, you can generate this skeleton function by executing the macro MakeCode.

In case one of the ntuple columns has a variable length (eg. px(ntrack)), h.Draw("px") will histogram the px column for all tracks in the same histogram. h.Draw("px[0]") is not implemented in version 0.90. Use the macro quoted above to generate the skeleton function and create/fill the relevant histogram yourself.

Accessing ROOT Files Remotely via a rootd Server

Reading and writing ROOTfiles over the net can be done by creating a `TNetFile` object instead of a `TFile` object. Since the `TNetFile` class inherits from the `TFile` class it has exactly the same interface and behaviour. The only difference is that it reads and writes to a remote `rootd` daemon.

TNetFile URL

`TNetFile` file names are in standard URL format with protocol "`root`". The following are valid `TNetFile` URL's:

```
root://hpsalo/files/aap.root
root://hpbrun.cern.ch/root/hsimple.root
root://pcna49a:5151/~na49/data/run821.root
root://pcna49d.cern.ch:5050//v1/data/run810.root
```

The only difference with the well known `httpd` URL's is that the root of the remote file tree is the remote user's home directory. Therefore an absolute pathname requires a `//` after the host or port specifier (as shown in the last example above). Further the expansion of the standard shell characters, like `~`, `$`, `..`, etc. is handled as expected. The default port on which the remote `rootd` listens is 432 and this default port is assumed by `TNetFile` (actually by `TUrl` which is used by `TNetFile`).

Remote Authentication

Connecting to a `rootd` daemon requires a remote user id and password. `TNetFile` supports three ways for you to provide your login information:

1. Setting it globally via the static `TNetFile` functions `TNetFile::SetUser()` and `TNetFile::SetPasswd()`
2. Via the `~/.netrc` file (same format and file as used by ftp)
3. Via command line prompt

The different methods will be tried in the order given above. On machines with AFS `rootd` will obtain an AFS token.

Using the General TFile::Open() Function

To make life simple we provide a general function to open any type of file (except shared memory files of class `TMapFile`). This functionality is provided by the static `TFile::Open()` function:

```
TFile *TFile::Open(const Text_t *name, Option_t *option="", const Text_t *title="",
```

Depending on the `name` argument the function returns either a `TFile`, a `TNetFile` or a `TWebFile` object. In case a `TNetFile` URL specifies a local file a `TFile` object will be returned (and of course no login information is needed). The arguments of the `Open()` function are the same as the ones for the `TFile` constructor.

A Simple Session

```
root [0] TFile *f1 = TFile::Open("local/file.root", "update")
root [1] TFile *f2 = TFile::Open("root://pcna49a.cern.ch/data/file.root", "new")
Name (pcna49a:rdm):
Password:
root [2] TFile *f3 = TFile::Open("http://root.cern.ch/~rdm/hsimple.root")
root [3] f3.ls() TWebFile** http://root.cern.ch/~rdm/hsimple.root TWebFile* http://root.cern.ch/~rdm/
hsimple.root
KEY: TH1F hpx;1 This is the px distribution
KEY: TH2F hpxpy;1 py vs px KEY: TProfile hprof;1 Profile of pz versus px
KEY: TNtuple ntuple;1 Demo ntuple
root [2] hpx.Draw()
```

The rootd Daemon

The `rootd` daemon works with the `TNetFile` class. It allows remote access to ROOT database files in either read or read/write mode. The `rootd` daemon can be found in the directory `$ROOTSYS/bin`. It can be started either via `inetd` or by hand from the command line (no need to be super user). Its performance is comparable with NFS but while NFS requires all kind of system permissions to setup, `rootd` can be started by any user. The simplest way to start `rootd` is by starting it from the command line while being logged in to the remote machine. Once started `rootd` goes immediately in the background (no need for the `&`) and you can log out from the remote node. The only argument required is the port number (> 1024) on which your private `rootd` will listen. Using `TNetFile` you can now read and write files on the remote machine. For example:

```
hpsalo [12] telnet fsgi02.fnal.gov login: minuser Password: <fsgi02> rootd -p 5151 <fsgi02> exit hpsalo [13] root root [0] TFile *f = TFile::Open("root://fsgi02.fnal.gov:5151/file.root","new") Name (fsgi02.fnal.gov:rdm): minuser Password: root [1] f.ls()
```

In the above example `rootd` runs on the remote node under user id `minuser` and listens to port `5151`. When creating a `TNetFile` object you have to specify this same port number `5151` and use `minuser` (and corresponding password) as login id. When `rootd` is started in this way you can only login with the user id under which `rootd` was started on the remote machine. However, you can make many connections since the original `rootd` will fork (spawn) a new `rootd` that will service the requests from the `TNetFile`. The original `rootd` keeps listening on the specified port for other connections. Each time a `TNetFile` makes a connection it gets a new private `rootd` that will handle its requests. At the end of a root session when all `TNetFiles` are closed only the original `rootd` will stay alive ready to service future `TNetFiles`.

Starting rootd via inetd

If you expect to often connect via `TNetFile` to a remote machine it is more efficient to install `rootd` as a service of the `inetd` super daemon. In this way it is not necessary for each user to run a private `rootd`. However, this requires a one time modification of two system files (and super user priviliges to do so). Add to `/etc/services` the line:

```
rootd432/tcp
```

and to `/etc/inetd.conf` the line:

```
rootd stream tcp nowait root /usr/local/root/bin/rootd rootd -i
```

After these changes force `inetd` to reread its config file with "`kill -HUP <pid inetd>`".

Setup in this way it is not necessary to specify a port number in the URL given to `TNetFile`. `TNetFile` assumes the default port to be 432 as specified above in the `/etc/services` file.

Rootd Command Line Arguments

`Rootd` support the following arguments:

```
-isays we are started by inetd -pport#specifies port number to listen on -d levellevel of debug info written to syslogd 0 = no debug (default) 1 = minimum 2 = medium 3 = maximum
```

Reading ROOT Files via Apache Web Server

By adding one ROOTspecific module to your Apache web server you can distribute ROOT files to any ROOTuser. No need anymore for ftp'ing files containing (out of date) histograms or other objects. Your latest up-to-date results are always accessable to all your colleagues.

How to access ROOTfiles via a web server. Simple, just create a `TWebFile` object instead of a `TFile` object with a standard URL as file name. For example:

`root [0] TWebFile f("http://root.cern.ch/~rdm/hsimple.root") root [1] f.ls() TWebFile** http://root.cern.ch/~rdm/hsimple.root TWebFile* http://root.cern.ch/~rdm/hsimple.root KEY: TH1F hpx;1 This is the px distribution KEY: TH2F hpxpy;1 py vs px KEY: TProfile hprof;1 Profile of pz versus px KEY: TNtuple ntuple;1 Demo ntuple root [2] hpx.Draw()`

Since `TWebFile` inherits from `TFile` all `TFile` operations work as expected. However, due to the nature of a web server a `TWebFile` is a readonly file. A `TWebFile` is ideally suited to read relatively small objects (like histograms or other data analysis results). Although possible, you don't want to analyse large `TTree`'s via a `TWebFile`.

Here follows a step-by-step recipe for making your Apache 1.1 or 1.2 web server ROOTaware:

4. Go to your Apache source directory and add the file mod_root.c.
5. Add to the end of the `Configuration` file the line: `Module root_module mod_root.o`
6. Run the `Configure` script
7. Type `make`
8. Copy the new httpd to its expected place
9. Go to the `conf` directory and add at the end of the `srm.conf` file the line: `AddHandler root-action root`
10. Restart the httpd server

That's all.

The ROOT Collection Classes

Collections are a key feature of the ROOT system. Many, if not most, of the applications you write will use collections. If you have used polymorphic C++ collections before, some of this material will be review. However, much of this tutorial covers aspects of collections specific to the ROOT system. In this tutorial the following features will be demonstrated:

- How to create instances of collections
- The difference between lists, ordered collections, hashtables, maps, etc.
- How to add and remove elements of a collection
- How to search a collection for a specific element
- How to access and modify collection elements
- How to iterate over a collection
- How to manage memory for collections and collection elements
- How collection elements are tested for equality (`IsEqual()`)
- How collection elements are compared (`Compare()`) in case of sorted collections
- How collection elements are hashed (`Hash()`) in hash tables
- Understanding collections Using collections

Understanding Collections

A collection is a group of related objects. You will find it easier to manage a large number of items as a collection. For example, collections of points and lines might be managed by a graphics pad. A vertex will have a collection of tracks. A detector geometry contains collections of shapes, materials, rotation matrices and subdetectors. Collections act as flexible alternatives to traditional data structures of computer science such as arrays, lists, and trees.

Collections can be thought of as polymorphic containers that can contain different types of elements. For this release of the ROOT system, elements to be placed in collections must be instances of classes. These may be classes defined by you or provided by ROOT. Collection elements must be instances of classes descending from TObject. The dependence of collections on TObject may disappear in the future when all C++ compilers used with the ROOT system fully support templates. In the mean time, knowing the role TObject plays in collections can be helpful.

In general you don't need to worry about TObject. Many ROOT classes have TObject as an ancestor. In fact, collections themselves are descendants of TObject. This makes it possible for collections to contain other collections (subcollections) in a tree structure. Such trees are used in the ROOT system to implement components of the graphics system (graphics pads containing pads), geometries (detectors in detectors), etc.

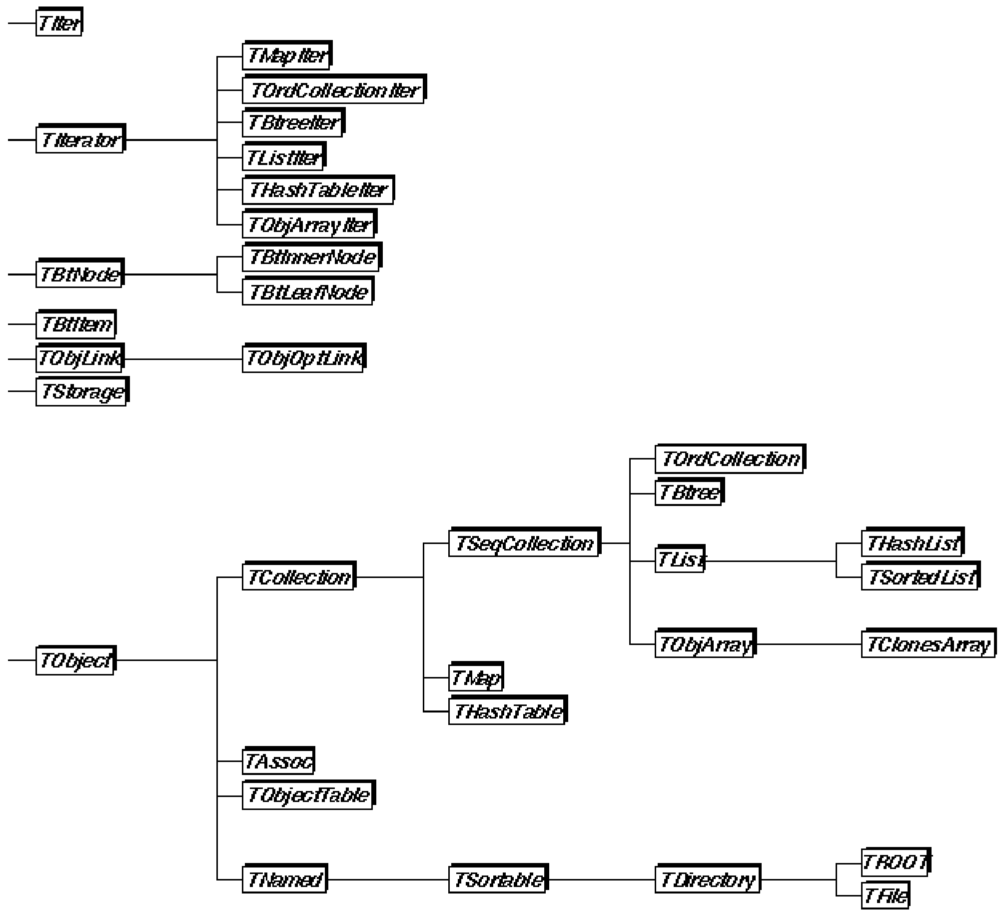
The basic protocol TObject defines for collection elements is shown below:

```
IsEqual()
Compare()
IsSortable()
Hash()
```

How to use and override these member functions is shown in the example program.

Types of Collections

The ROOT system implements the following type of collections: arrays, lists, sorted lists, B-trees, hashtables and maps. The figure below shows the inheritance hierarchy for the primary collection classes.



Ordered Collections (Sequences)

Sequences are collections that are externally ordered because they maintain internal elements according to the order in which they were added. The following sequence collections are available:

- TList
- THashList
- TOrdCollection
- TObjArray
- TClonesArray

Both a `TObjArray` as well as a `TOrdCollection` can be sorted using their `Sort()` member function (assuming the stored items are sortable).

Sorted Collections

Sorted collections are ordered by an internal (automatic) sorting mechanism. The following sorted collections are available:

- TSortedList
- TBtree

Unordered Collections

Unordered collections don’t maintain the order in which the elements were added. I.e. when you iterate over an unordered collection, you are not likely to retrieve elements in the same order they were added to the collection. The following unordered collections are available:

- THashTable
- TMap

Using Collection Classes

```
#include <stdlib.h>
#include <iostream.h>
#include "Root.h"
#include "RString.h"
#include "ObjString.h"
#include "SortedList.h"
#include "ObjArray.h"
#include "OrdCollection.h"
#include "HashTable.h"
// To focus on basic collection protocol, this sample program uses
// simple classes inheriting from TObject. One class, TObjString, is a
// collectable string class (a TString wrapped in a TObject) provided
// by the ROOT system. The other class we define below, is an integer
// wrapped in a TObject, just like TObjString.
// TObjNum is a simple container for an integer.
class TObjNum : public TObject {
private:
  int  num;
public:
  TObjNum(int i = 0) : num(i) { }
  ~TObjNum() { Printf("~TObjNum = %d", num); }
  void    SetNum(int i) { num = i; }
  int     GetNum() { return num; }
  void    Print(Option_t *) { Printf("TObjNum = %d", num); }
  ULong_t  Hash() { return num; }
  Bool_t IsEqual(TObject *obj) { return num == ((TObjNum*)obj)->num; }
  Bool_t  IsSortable() const { return kTRUE; }
  Int_t   Compare(TObject *obj) { if (num == ((TObjNum*)obj)->num)
                                     return 0;
                                  else if (num < ((TObjNum*)obj)->num)
                                     return -1;
                                  else
                                     return 1; }
};
// Initialize the ROOT framework
TROOT tcollection("Collection","Test collection classes");
int main()
{
  int i;
  Printf(
  "//////////////////////////////////////////////////////////////////
"
  "// Test of TObjArray                                          //
"
  "//////////////////////////////////////////////////////////////////"
  );
   // Array of capacity 10, Add() will automatically expand the array if necessary.
  TObjArray  a(10);
  Printf("Filling TObjArray");
  a.Add(new TObjNum(1));            // add at next free slot, pos 0
  a[1] = new TObjNum(2);            // use operator[], put at pos 1
  TObjNum *n3 = new TObjNum(3);
  a.AddAt(n3,2);                    // add at position 2
  a.Add(new TObjNum(4));            // add at next free slot, pos 3
  a.AddLast(new TObjNum(10));       // add at pos 4
  TObjNum n6(6);                    // stack based TObjNum
  a.AddAt(&n6,5);                   // add at pos 5
  a[6] = new TObjNum(5);            // add at respective positions
  a[7] = new TObjNum(8);
  a[8] = new TObjNum(7);
//   a[10] = &n6;                    // gives out-of-bound error
  Printf("Print array");
  a.Print();                        // invoke Print() of all objects
  Printf("Sort array");
  a.Sort();
  for (i = 0; i < a.Capacity(); i++)  // typical way of iterating over array
     if (a[i])
        a[i]->Print();      // can also use operator[] to access elements
     else
        Printf("%d empty slot", i);
  Printf("Use binary search to get position of number 6");
  Printf("6 is at position %d", a.BinarySearch(&n6));
  Printf("Find number before 6");
  a.Before(&n6)->Print();
  Printf("Find number after 3");
  a.After(n3)->Print();
  Printf("Remove 3 and print list again");
  a.Remove(n3);
  delete n3;
  a.Print();
  Printf("Iterate forward over list and remove 4 and 7");
   // TIter encapsulates the actual class iterator. The type of iterator
```

```
    // used depends on the type of the collection.
   TIter next(&a);
   TObjNum *obj;
   while (obj = (TObjNum*)next())     // iterator skips empty slots
      if (obj->GetNum() == 4) {
         a.Remove(obj);
         delete obj;
      }
    // Reset the iterator and loop again
   next.Reset();
   while (obj = (TObjNum*)next())
      if (obj->GetNum() == 7) {
         a.Remove(obj);
         delete obj;
      }
   Printf("Iterate backward over list and remove 2");
   TIter next1(&a, kIterBackward);
   while (obj = (TObjNum*)next1())
      if (obj->GetNum() == 2) {
         a.Remove(obj);
         delete obj;
      }
   Printf("Delete remainder of list: 1,5,8,10 (6 not deleted since not on heap)");
    // Delete heap objects and clear list. Attention: do this only when you
    // own all objects stored in the collection. When you stored aliases to
    // the actual objects (i.e. you did not create the objects) use Clear()
    // instead.
   a.Delete();

   Printf(
   "////////////////////////////////////////////////////////////////
"
   "// Test of TOrdCollection                                    //
"
   "////////////////////////////////////////////////////////////////"
   );
    // Create collection with default size, Add() will automatically expand
    // the collection if necessary.
   TOrdCollection  c;
   Printf("Filling TOrdCollection");
   c.Add(new TObjString("anton"));      // add at next free slot, pos 0
   c.AddFirst(new TObjString("bobo"));  // put at pos 0, bump anton to pos 1
   TObjString *s3 = new TObjString("damon");
   c.AddAt(s3,1);                       // add at position 1, bump anton to pos 2
   c.Add(new TObjString("cassius"));    // add at next free slot, pos 3
   c.AddLast(new TObjString("enigma")); // add at pos 4
   TObjString s6("fons");               // stack based TObjString
   c.AddBefore(s3,&s6);                 // add at pos 1
   c.AddAfter(s3, new TObjString("gaia"));
   Printf("Print collection");
   c.Print();                           // invoke Print() of all objects
   Printf("Sort collection");
   c.Sort();
   c.Print();
   Printf("Use binary search to get position of string damon");
   Printf("damon is at position %d", c.BinarySearch(s3));
   Printf("Find str before fons");
   c.Before(&s6)->Print();
   Printf("Find string after damon");
   c.After(s3)->Print();
   Printf("Remove damon and print list again");
   c.Remove(s3);
   delete s3;
   c.Print();
   Printf("Iterate forward over list and remove cassius");
   TObjString *objs;
   TIter next2(&c);
   while (objs = (TObjString*)next2())     // iterator skips empty slots
      if (objs->String() == "cassius") {
         c.Remove(objs);
         delete objs;
      }
   Printf("Iterate backward over list and remove gaia");
   TIter next3(&c, kIterBackward);
   while (objs = (TObjString*)next3())
      if (objs->String() == "gaia") {
         c.Remove(objs);
         delete objs;
      }
   Printf("Delete remainder of list: anton,bobo,enigma (fons not deleted since not on heap)");
   c.Delete();                         // delete heap objects and clear list
   Printf(
   "////////////////////////////////////////////////////////////////
"
   "// Test of TList                                             //
"
   "////////////////////////////////////////////////////////////////"
```

```
   );
    // Create a doubly linked list.
   TList l;
   Printf("Filling TList");
   n3 = new TObjNum(3);
   l.Add(n3);
   l.AddBefore(n3, new TObjNum(5));
   l.AddAfter(n3, new TObjNum(2));
   l.Add(new TObjNum(1));
   l.AddBefore(n3, new TObjNum(4));
   l.AddFirst(&n6);
   Printf("Print list");
   l.Print();
   Printf("Remove 3 and print list again");
   l.Remove(n3);
   delete n3;
   l.Print();
   Printf("Iterate forward over list and remove 4");
   TIter next4(&l);
   while (obj = (TObjNum*)next4())
      if (obj->GetNum() == 4) l.Remove(obj);
   Printf("Iterate backward over list and remove 2");
   TIter next5(&l, kIterBackward);
   while (obj = (TObjNum*)next5())
      if (obj->GetNum() == 2) {
         l.Remove(obj);
         delete obj;
      }
   Printf("Delete remainder of list: 1, 5 (6 not deleted since not on heap)");
   l.Delete();
   Printf(
   "////////////////////////////////////////////////////////////////
"
   "// Test of TSortedList                                        //
"
   "////////////////////////////////////////////////////////////////"
   );
    // Create a sorted doubly linked list.
   TSortedList sl;
   Printf("Filling TSortedList");
   n3 = new TObjNum(3);
   sl.Add(n3);
   sl.AddBefore(n3,new TObjNum(5));
   sl.AddAfter(n3, new TObjNum(2));
   sl.Add(new TObjNum(1));
   sl.AddBefore(n3, new TObjNum(4));
   sl.AddFirst(&n6);
   Printf("Print list");
   sl.Print();
   sl.Delete();
   Printf(
   "////////////////////////////////////////////////////////////////
"
   "// Test of THashTable                                         //
"
   "////////////////////////////////////////////////////////////////"
   );
    // Create a hash table with an initial size of 20 (actually the next prime
    // above 20). No automatic rehashing.
   THashTable ht(20);
   Printf("Filling THashTable");
   Printf("Number of slots before filling: %d", ht.Capacity());
   for (i = 0; i < 1000; i++)
      ht.Add(new TObject);
   Printf("Average collisions: %f", ht.AverageCollisions());
    // rehash the hash table to reduce the collission rate
   ht.Rehash(ht.GetSize());
   Printf("Number of slots after rehash: %d", ht.Capacity());
   Printf("Average collisions after rehash: %f", ht.AverageCollisions());
   ht.Delete();
    // Create a hash table and trigger automatic rehashing when average
    // collision rate becomes larger than 5.
   THashTable ht2(20,5);
   Printf("Filling THashTable with automatic rehash when AverageCollisions>5");
   Printf("Number of slots before filling: %d", ht2.Capacity());
   for (i = 0; i < 1000; i++)
      ht2.Add(new TObject);
   Printf("Number of slots after filling: %d", ht2.Capacity());
   Printf("Average collisions: %f", ht2.AverageCollisions());
   ht2.Delete();

   Printf("End of program, dtor's do the rest");

   return 0;
}
```

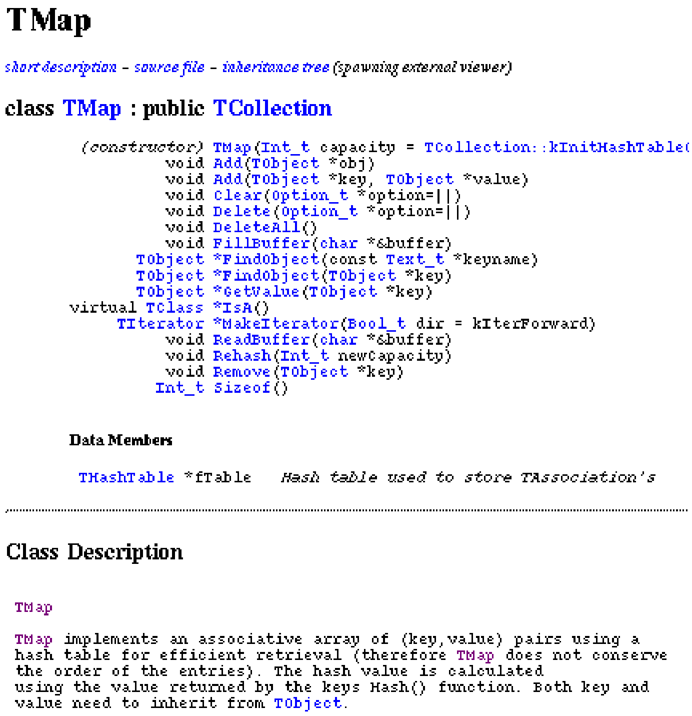
Automatic Document Generation

Save Enormous Amount Of Time

Developers are an expensive resource and documenting the code manually requires too much precious developer time, also it's a daunting task and very often postponed or even avoided.

Automatic Document Generation saves developers enormous amounts of time and effort by providing instant, detailed and always up to date information on the internals of the code.

Using **ROOT** it is possible to automatically and continuously update all aspects of the source code documentation, so that the entire team has the all necessary information at their fingertips. Using the information stored in the dictionary and the source files **ROOT** can automatically generate a source code documentation both in **HTML** and **PostScript®** format.



TMap

short description - source file - inheritance tree (spawning external viewer)

class TMap : public TCollection

```
(constructor) TMap(Int_t capacity = TCollection::kInitHashTable(
         void Add(TObject *obj)
         void Add(TObject *key, TObject *value)
         void Clear(Option_t *option=||)
         void Delete(Option_t *option=||)
         void DeleteAll()
         void FillBuffer(char *&buffer)
      TObject *FindObject(const Text_t *keyname)
      TObject *FindObject(TObject *key)
      TObject *GetValue(TObject *key)
virtual TClass *IsA()
    TIterator *MakeIterator(Bool_t dir = kIterForward)
         void ReadBuffer(char *&buffer)
         void Rehash(Int_t newCapacity)
         void Remove(TObject *key)
        Int_t Sizeof()
```

Data Members

```
THashTable *fTable   Hash table used to store TAssociation's
```

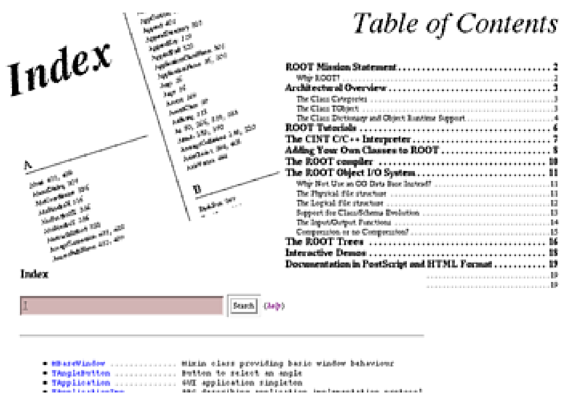
Class Description

```
TMap

TMap implements an associative array of (key,value) pairs using a
hash table for efficient retrieval (therefore TMap does not conserve
the order of the entries). The hash value is calculated
using the value returned by the keys Hash() function. Both key and
value need to inherit from TObject.
```

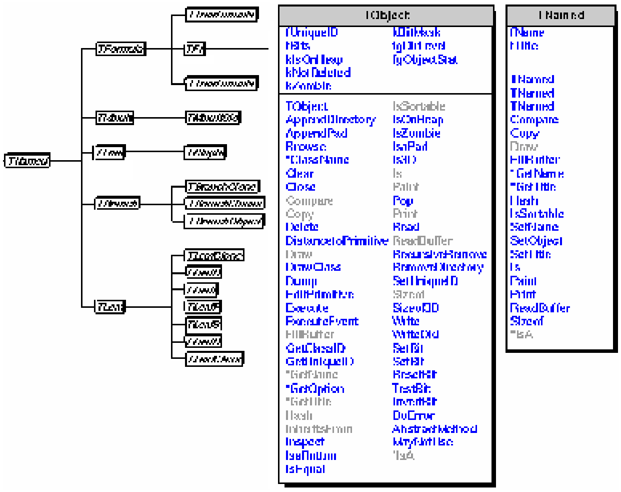
Navigate Quickly

Because of the vast amounts of information in the source code it is often difficult to find the right detail just when you want it. Through *highly structured documents*, *indexed* and *hypertext linked*, developers can navigate quickly and easily to the information that matters. Locating and investigating the code is easier than ever, and of course code reuse too. High quality, indexed and on-line documentation is key to success.



Visualize The Code

Many prefer to visualize the code, so we provide a graphical representation of *inheritance diagrams*. In capturing the essence of the code and organizing it logically, **ROOT** improves the development process significantly with no incremental developer effort!



Requirements

You need **comments** placed at the **three strategic points** in your source code.

Comments behind a data member definition are stored in the dictionary like a **data member description** and they are used whenever data member needs to be described..

```
class TMap : public TCollection {

   private:
      THashTable    *fTable;    //Hash table used to store TAssociation's
       ...    ...    ...    ...    ...    ...    ...
       ...    ...    ...    ...    ...    ...    ...
}
```

The first comment lines after the opening bracket ({) of a member function are considered as a **member function description**.

```
void TMap::Delete(Option_t *option)
{
// Remove all (key,value) pairs from the map AND delete the keys
// when they are allocated on the heap.

   ...    ...
   ...    ...
}
```

A commented block at the top of the source file is considered as a **class description**.

```
//////////////////////////////////////////////////////////////////////////
//                                                                      //
// TMap                                                                 //
//                                                                      //
// TMap implements an associative array of (key,value) pairs using a    //
// hash table for efficient retrieval (therefore TMap does not conserve //
// the order of the entries). The hash value is calculated              //
// using the value returned by the keys Hash() function. Both key and   //
// value need to inherit from TObject.                                  //
//                                                                      //
//////////////////////////////////////////////////////////////////////////
```

How To Start?

First of all you have to be sure that your class (or classes) are properly integrated into ROOT. If that's the case then you can expect no problems at all (of course with a little bit of luck). Here is the small example showing how to create your first HTMLfile:

```
1    root> THtml html;
2    root> html.SetOutputDir( "/tmp/doc/" );
3    root> html.SetSourceDir( "./source/cpp/" );
4    root> html.MakeClass( "MyClass" );
5                  /tmp/doc/MyClass.html
6                  /tmp/doc/MyClass_Tree.ps
```

As you can see the first thing to do is to create an object of type THtml *(line 1)*. Then we need to decide where to put output files *(line 2)*, but keep in mind that selected **output directory must exist** and if it doesn't then you have to create one.The third thing is to give a path to your source directories *(line 3)* , which means that in case you have separate directories for source files (.c, .cpp, .C) and include files (.h .H) your command should look like:

```
3    root> html->SetSourceDir( "source/cpp/:source/h/" );
```

Finally *(line 4)*, you are ready to *make a class.* Your screen output should match the output shown *(lines 5 & 6)*.

What Else Should I Know?

Very likely you have written more than one class, and now you want to generate HTML files for all of them. Of course you can follow the same steps you've learned in previous chapter for each of your classes, or you can write a small macro doing it, but using ROOT you have a possibility to *make them all* at once. Here is the example:

```
root> THtml html;
root> html2.SetOutputDir( "/tmp/doc/" );
root> html2.SetSourceDir( "./source/cpp/" );
root> html2.MakeAll( );
  -no change-   /tmp/doc/MyClass.html
  -no change-   /tmp/doc/MyClass_Tree.ps
                /tmp/doc/MySecondClass.html
                /tmp/doc/MySecondClass_Tree.ps
```

Did you notice that the interface is the same? Great, that's why is so easy. But why we have "*-no change-*" messages this time? Because, make will look for modifications and it will regenerate files modified since last time only, but if you want to force it to generate all files then you can do it this way:

```
root> html.MakeAll( kTRUE );\
```

Advanced Configuration

Are you tired setting an output directory and a source path each time you want to run HTML utility? Here is the solution. You have to add two special lines in your .rootrc configuration file.

```
Root.Html.OutputDir: /home/rootuser/src/html
Root.Html.SourceDir: /home/rootuser/src:/home/rootuser/include
```

Also there is a way to tune ROOT even further. It's always good to know the author's name to whom certain class belongs to. This is possible, but you need to define a format for author's signature used in your source code, which actually means that, for the time being, you have to specify a leading string in front of the author's name. Here is the example:

```
Root.Html.Author: //----------Author's Name:
```

In this case ROOT will try to match any line starting with that pattern and everything else in that line will be displayed as author's name, i.e. one correct line should be:

```
//----------Author's Name:  Barton Fink, 1991
```

It's a same thing for showing your copyright notice or displaying last modified info.

```
Root.Html.LastUpdate: //----------Modified:
Root.Html.Copyright:  //----------Copyright:
```

For additional info, see THtml class description.

René Brun, Nenad Buncic, Fons Rademakers

Last update 14/01/97 by NB

C++ Coding Conventions

Naming conventions

For naming conventions we follow the Taligent rules. Up to now they have written the largest body of C++ and the rules seem well thought out. No need to invent something else. The only addition/change we made is to append an `_t` to typedefs and simple structs, e.g.:

```
typedef int Int_t;
struct Simple_t { ..... };
```

Addherence to the rules is mandatory. Any deviation will cause confusion and chaos. After a while one really gets used to the fact that all class fields start with an `f` followed by a capitalized word, `fEnergy`, or that `TCollection` is a class. If the convention is sporadically violated debugging becomes a nightmare. The usage of a standard begin letter or token for the different types also makes it easy to parse and search the code using simple tools.

Class definition conventions

Also here the Taligent guide is quite reasonable. Of course, no class data member should ever be public (OO remember). Make the data fields always private. Or protected, if you want to grant an inherited class direct access. Concerning inlines. Add trivial get or setters directly in the class definition (improves reading time since one does not have to look for it somewhere else). Add more complex inlines (longer than one line) at the bottom of the `.h` file. Creating separate `.icc` files increase the build time, the complexity of the `Makefile` and, more importantly, increases the number of files one possibly has to scan to find a piece of code (remember code is write once, read often).

Avoid raw C types

Avoid the use of raw C types like `int`, `long`, `float`, `double` when using data that might be written to disk. For example, the sizes of `int` and `long` are machine dependent. On an old PC an `int` is 16 bits and a `long` is 32 bits. On 32 bit workstations `int`'s are 32 bits, but on 64 bit processors an `int` can be either 32 or 64 bits, depending on the processor. For portability reasons and consistent numerical results use typedefs for the basic raw C types. E.g.:

```
#ifdef B64
typedef int            Long_t;      //large integer
typedef unsigned int   ULong_t;     //large integer
#else
typedef long           Long_t;      //large integer
typedef unsigned long  ULong_t;     //large integer
#endif
```

Click here to view all ROOT defined types.

New features like, templates, exception handling, RTTI and namespaces

Of these four new language features templates are the most evolved. However, good template support is still very limited and very compiler dependent. Most Cfront based compilers take extremely long instantiating templates when linking the program. One or two template classes are no problem, but more causes an exponential degradation in link performance. This is especially painful in the frequent edit-compile-link cycles during code development. Some PC compilers have better template support. In general it is expected to take a few more years before templates are performantly supported over the full range of platforms. In the meantime, design your classes in such a way that later they can easily be replaced by templatized versions.

Exception handling is only rudimentary supported by most compilers. Here, however, no link time penalty but a run time penalty is incured (the run time needs to keep track of the resources to be freed when an exception is thrown). Also here PC compilers are leading the way. For the time being wrap setjmp and longjmp calls in TRY, THROW and CATCH macros. This will not provide the full exception handling functionality but at least it will get you going and enables your program to handle the most basic problems like floating point exceptions, bus errors, segmentation violations, etc.

RTTI, run time type identification, is the latest addition to the proposed C++ standard. Only a few PC compilers support it. Once RTTI is allround available it will allow a simplification of many I/O schemes (this is also one of the reasons why the current ODMG standardization effort is way too premature (who would have liked to standardize C++ in 1985?)).

Namespaces are not even supported by PC compilers. As long as they cannot be used extend the `T` for class names to something like `TNa...`, e.g. `TNaTrack`, for NA49 code.

Using comments to document the code

Using the information stored in the dictionary and the source files ROOT can automatically generate a hyperized version of the header and source files. By placing comments in a few strategic places a complete reference manual, including graphics, can be generated using the `THtml` class.

There are three different kind of comments used by `THtml`.

Data member description comments

Comments behind a data member definition in the header file, e.g.:

```
  Float_t     fEnergy;     // the particle energy
```

The comment "`the particle energy`" will be put in the dictionary and used whenever `fEnergy` needs to be described (e.g. on-line help). Have a look at a raw header file and compare it to the hyperized version.

Class description comments

A comment block at the top of the source file describing the class. This block has to be preceeded by a line containing a C++ comment token followed a delimiter string. All comment lines after this starting line are part of the class description, e.g.:

```
//______________________________________________________________________
//
// TPrimitive
//
// This class is the abstract base class of geometric primitives.
// Use this class by inheriting from it and overriding the
// members, Draw() and List().
//
```

For an example of a class description comment block see this source file and compare it to the hyperized version.

Member function description comments

The first comment lines after the opening bracket (`{`) of a member function are considered a description of the function. For example:

```
TList::Insert()
{
  // Insert node into linked list.
  // To insert node at end of list use Add().
  ...
  ...
}
```

Here the two comment lines after the `{` are used to describe the working and usage of the member function. For a real life example see this source file and compare it to the hyperized version.

HTML directly in the source

In the class and member function description comments one can also insert "raw" HTML. This gives the possibility to have nicely formatted text or, more importantly, to be able to include graphics into the source showing, for example, a data structure, an algorithm an inheritance tree, etc. For example:

```
TList::Insert()
{
  // Insert node into linked list.
  // To insert node at end of list use Add().
  //Begin_Html
  /*
  */
  //End_Html
  ...
  ...
}
```

Everything enclosed by the `//Begin_Html` and `//End_Html` lines is considered to be pure HTML.

ROOT Input/Output Benchmarks

The table below (also available in Postscript format) shows the result of a ROOT benchmark run on a Pentium Pro 200 Mhz (HP Vectra XU) running **RedHat Linux v4.0** (kernel 2.0.27). The machine has 256 Mbytes of RAM and I/O is on the local disk (Seagate Barracuda 9GB Ultra SCSI).

The standard test program Event has been used to generate the data file in various configurations with 1000 events for each configuration. The data file is organized in TTree format. The program was invoked with:

Event 1000 comp split

where:

• comp = 0 means: no compression at all.

• comp = 1 means: compress everything if split = 0.

• comp = 1 means: compress only the **Tree** branches with integers if split = 1.

• comp = 2 means: compress everything if split=1.

• split = 0 : the full event is serialized into one single buffer.

• split = 1 : the event is split into branches. One branch for each data member of the **Event** class. The list of tracks (a TClonesArray) has the data members of the **Track** class also split into individual buffers.

The **Total Time** column is the real time in seconds to run the program. **Effective time** corresponds to the real time minus the time spent in non I/O operations (essentially the random number generator).

The program Event generates in average 600 tracks per event. Each track has 17 data members.

The read benchmark runs under the interactive version of ROOT. The **Total time to read All** is the real time reported by the execution of the macro eventa, via the ROOT interpreter. We did not correct this time for the overhead coming from the interpreter itself.

The **Total time to read Sample** corresponds to the execution of the macro eventb. This macro loops on all events. For each event, the branch containing the number of tracks is read. In case the number of tracks is less than 585, the full event is read in memory. This test is obviously not possible in non-split mode. In non-split mode, the full event must be read in memory.

Event Parameters	File Size (Comp. factor)	Total time to write (Mbytes/s)	Effective time to write (Mbytes/s)	Total time to read All (Mbytes/s)	Total time to read Sample (Mbytes/s)
Comp = 0 Split = 0	48.7 Mbytes (1.0)	19.85s (2.44 MB/s)	7.35s (6.62 MB/s)	7.87s (6.17 MB/s)	not possible
Comp = 0 Split = 1	47.15 Mbytes (1.0)	21.25s (2.12 MB/s)	8.75s (5.39 MB/s)	8.39s (5.38 MB/s)	1.40s (31.77 MB/s)
Comp = 1 Split = 1	36.85 Mbytes (1.30)	24.97s (1.81 MB/s)	12.47s (3.78 MB/s)	8.69s (5.19 MB/s)	1.69s (26.3 MB/s)
Comp = 1 Split = 0	32.01 Mbytes (1.52)	67.02s (0.72 MB/s)	54.52s (0.89 MB/s)	17.07s (2.84 MB/s)	not possible
Comp = 2 Split = 1	27.20 Mbytes (1.73)	65.4s (0.69 MB/s)	52.9s (0.89 MB/s)	16.26s (2.78 MB/s)	4.2s (10.6 MB/s)

This benchmark illustrates the pros and cons of the compression option. Use compression where the time spent in I/O is small compared to the total processing time or if the compression factor is high.

Note that, so far, we did not make any special effort to tune the ROOT I/O performance. We expect to improve this area in the coming releases.

The times reported in the table correspond to complete I/O operations necessary to deal with **machine independent binary files**. On **Linux**, this also includes byte-swapping operations. The ROOT file allows for direct access to any event in the file and also direct access to any part of an event when split=1.

Note also that the uncompressed file generated with split=0 is 48.7 Mbytes and only 47.17 Mbytes for the option split=1. The difference in size is due to the object identification mechanism overhead when the event is written to a single buffer. This overhead does not exist in split mode because the branch buffers are optimized for homogeneous data types.

You can run yourself the test programs on your architecture. The program Event will report the write performance. You can measure the read performance by executing the macros eventa and eventb. The performance depends not only of the processor type, but also of the disk devices (local, NFS, AFS, etc.).

ROOT compared to ZEBRA and HBOOK

The table below (also available in Postscript format) shows the result of a ROOT benchmark run on a Pentium Pro 200 Mhz (HP Vectra XU) running **RedHat Linux v4.0** (kernel 2.0.27). The machine has 256 Mbytes of RAM and I/O is on the local disk (Seagate Barracuda 9GB Ultra SCSI).

The standard test program Event has been used to generate the ROOT data file. Results of running this program in many different configurations are presented in ROOT I/O benchmarks. This program generates 10,000 events. Each event has an average of 600 tracks. Eack track has 17 data members. In addition each event includes an event-header object and an histogram object. The program **Event** was run with the option no compression and no split in order to compare with the only possibility offered by the ZEBRA Input/Output system. This program has been run in different configurations selected by a program argument to test the amount of time spent in creating and filling the event/track objects, the time to fill 20 histograms (about 120,000,000 fills). The time to write the file is computed as the total real time to run the program minus the time to create the objects (essentially the random number generator). The macro eventa is used to compute the total real time to read this file. Reading is via the interpreter. Both the ROOT file and ZEBRA files are written in a machine independent format. The ROOT library and the test program were compiled with the default level of optimisation.

A Fortran program zEvent has been written to perform exactly the same tasks as the ROOT C++ program. This program creates a top level ZEBRA bank lEvent with 3 down links: one bank to describe the event header, one bank to contain one histogram, one bank with a number of down links corresponding to the number of tracks per event. Each track bank has 17 data members. Histograms are created and filled using the HBOOK histogram package. The program includes options to compute the total time in creating and filling the ZEBRA data structures, the time to fill histograms and the total I/O time. Both the sequential I/O via FZ and direct access via RZ have been investigated.

	ROOT	Zebra RZ	Zebra FZ
Time to Create Structures	119s	137s	137s
Time to Fill Histograms	130s	200s	200s
Real Time to Write	81s	146s	116s
Real Time to Read	75s	103s	89s
File Size (Mbytes)	482	679	679

From this table, we see that:

• ROOT is 1.63 times faster than HBOOK to fill histograms. The time to fill histograms with ROOT is independent of the number of histograms. With HBOOK, there is a log(n) dependency. By default the ROOT histogram filling functions always increment in double precision the quantities to compute correctly the mean value and standard deviations. If the similar option is activated in HBOOK, then the filling time increases from 206 seconds to 280 seconds. In reality, ROOT is at least 2 times faster than HBOOK to fill histograms.

• ROOT can **write or read more than 100,000 objects per second**.

• ROOT is 1.89 times faster than ZebraRZ to write.

• ROOT is 2.99 times faster than ZebraFZ to write.

• ROOT is 1.24 times faster than ZebraRZ to read.

• ROOT is 1.10 times faster than ZebraFZ to read.

• The Zebra file is 1.40 times bigger than the ROOT file.

• If the ROOT program is run with the file compression option for branches with integers and the split method, the size of the file is 368 Mbytes (1.84 times smaller than ZEBRA) and the time spent in writing is 128 seconds. Note that this mode is particularly interesting because it offers **direct-access to any event** in the file and

direct-access to subsets of one event.

• If the ROOT program is run with the file compression option and the serialization method the size of the file is 271 Mbytes (2.50 times smaller than ZEBRA), but the time spent in writing is 605 seconds.

Note that this test has been done with a very large file to avoid the system mapping the file into memory. A test with only 1000 events shows that ROOT can read 1000 events in 3.4 seconds when the file is mapped to memory. This corresponds to 13 Mbytes per real time second.

The table below (also available in Postscript format) shows the results when running on the same machine with the same hardware configuration and under the Windows/NT operating system. In this case, Microsoft VisualC++ is used for ROOT and CERNLIB96a compiled with Microsoft Fortran for the ZEBRA and HBOOK test. The ROOT library and the test program were compiled with the maximum level of optimisation. The very bad performance of CERNLIB96a is under investigation.

	ROOT	Zebra RZ	Zebra FZ
Time to Create Structures	90s	222s	222s
Time to Fill Histograms	91s	878s	878s
Real Time to Write	99s	211s	122s
Real Time to Read	73s	215s	157s
File Size (Mbytes)	482	679	679

The table below (also available in Postscript format) shows the results when running on a Pentium PRO 200 Mhz with 128 Mbytes of memory running the SOLARIS operating system. The machine has an Adaptec 2940 Ultra Wide SCSI adapter. The disk used is 1 Gbyte Seagate Hawk 2XL Wide. In this case, SUN C++ is used for ROOT and SUN Fortran77 for the ZEBRA and HBOOK test (CERNLIB96a). The ROOT library and the test program were compiled with the default level of optimisation.

	ROOT	Zebra RZ	Zebra FZ
Time to Create Structures	194s	176s	176s
Time to Fill Histograms	144s	264s	264s
Real Time to Write	86s	124s	108s
Real Time to Read	170s	260s	284s
File Size (Mbytes)	482	679	679

Status & Availability

Development Status

The ROOT system is developed using the CMZ code management system. The ROOT source code is contained in one single CMZ file **root.cmz**. This file (about 3.6 Mbytes) is structured in several directories corresponding to the class categories of the system. An automatic installation procedure for all Unix systems and Windows/NT is provided in the **$KUMACS** directory together with other utilities. This structure has greatly facilitated our development procedure in an heterogeneous environment. The history of the development is kept in the file itself.

• Version 1.00/10 First Public Release (25/4/97)

• Version 1.02/00 Second Public Release (28/7/97)

• Version 1.03/07 Development version (8/11/97)

Porting Status

The ROOT system consists of about 370,000 lines of code (278,000 lines C++ and 92,000 lines C). The C language is used in CINT and in pieces of public domain code that perform specific functions like, terminal I/O handling (Getline), data compression (Zip) and the 3D interactive interface to X/Windows (X3D).

ROOT is currently running on the following platforms:

• Linux (g++)

• HP-UX 9.x and 10.x (HP CC and aCC C++ compilers)

• IBM AIX 4.1 and 3.2 (xlc compiler)

• Sun/Solaris for SPARC (SUN C++ compiler)

• Sun/Solaris for Intel PCs (SUN C++ compiler)

• DEC/Alpha/Unix (g++ and DEC/CXX)

• SGI/Irix (g++ and SGI C++ compiler)

• Windows NT and Windows95 (Visual C++ compiler)

• Mac under MkLinux (g++)

• MacOS (CodeWarrior, no graphics)

ROOT can be compiled without problems by the GNU g++ compiler on most Unix platforms.

Availability

Via Anonymous FTP

The following versions of the ROOT system are available via anonymous ftp:

• Version 1.02/10 Second Public Release. See Release notes for more details.

• Version 1.03/07 Development version: RECOMMENDED. See Development notes for more details.

ATTENTION: whenever running on a machine which is an AFS client, make sure your library path environment variable is set properly as described in the next section. If not you might be accessing the ROOTshared libraries from CERN resulting in a severe performance degradation.

Documentation in PostScript and HTML Format

PostScript Documentation

The following PostScript files have been generated by automatically scanning the ROOT HMTL files. This documentation includes page numbers, table of contents and an index.

• ROOT Overview: Overview of the ROOT system (361 KB, 81 pages)

• ROOT Tutorials: The ROOT tutorials with graphics examples (320 KB, 81 pages)

• ROOT Classes: Description of all the ROOT classes (1.46 MB, 661 pages)

HTML Documentation

In case you only have access to a low-speed connection to CERN, you can get a copy of the complete ROOT html tree (13.2 MB). All links are relative links to the top of this tree.

Like for the PostScript files, please note that we are frequently updating our Web pages. **These files refer to version 1.02. and have been generated on August 14 1997**.

Setting the Library Path

On **HP-UX**, before executing the interactive module, you must set the library path:

export SHLIB_PATH=$SHLIB_PATH:$ROOTSYS/lib (in bash or ksh) setenv SHLIB_PATH ${SHLIB_PATH}:${ROOTSYS}/lib (in csh or tcsh)

On **AIX**, before executing the interactive module, you must set the library path:

[-z "$LIBPATH"] && export LIBPATH=/lib:/usr/lib export LIBPATH=$LIBPATH:$ROOTSYS/lib

or the equivalent setenv.

On **Linux**, **Solaris**, **Alpha OSF** and **SGI**, before executing the interactive module, you must set the library path:

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ROOTSYS/lib

or the equivalent setenv.

On **Solaris**, in case your LD_LIBRARY_PATH is empty, you should set it like this:

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$ROOTSYS/lib:/usr/dt/lib

where **$ROOTSYS** is an environment variable pointing to the ROOT directory. For example, if you use the HPUX-10 AFS version (see below), you should set:

```
export ROOTSYS=/afs/cern.ch/na49/library/local/ROOT/hpux10/root(in bash or ksh)
```

setenv ROOTSYS /afs/cern.ch/na49/library/local/ROOT/hpux10/root(in csh or tcsh)

To run the program just type: `root`.

Via AFS

An executable version of ROOT for Linux, HP-UX and AIX is available on afs at the location:

/afs/cern.ch/na49/library/local/ROOT/i386_linux2/root/bin/root /afs/cern.ch/na49/library/local/ROOT/hpux9/root/bin/root /afs/cern.ch/na49/library/local/ROOT/hpux10/root/bin/root /afs/cern.ch/na49/library/local/ROOT/aix4/root/bin/root

We also provide the ROOT libraries:

/afs/cern.ch/na49/library/local/ROOT/i386_linux2/root/lib/*.sl /afs/cern.ch/na49/library/local/ROOT/hpux9/root/lib/*.sl /afs/cern.ch/na49/library/local/ROOT/hpux10/root/lib/*.sl /afs/cern.ch/na49/library/local/ROOT/aix4/root/lib/libRoot.a and libCint.a

and the include files:

/afs/cern.ch/na49/library/local/ROOT/i386_linux2/root/include/*.h /afs/cern.ch/na49/library/local/ROOT/hpux9/root/include/*.h /afs/cern.ch/na49/library/local/ROOT/hpux10/root/include/*.h /afs/cern.ch/na49/library/local/ROOT/aix4/root/include/*.h

Demo macros can be found in the tutorial section. The source of these macros is available under afs at the location:

/afs/cern.ch/na49/library/local/ROOT/hpux10/root/tutorials/*.C and *.root

Examples of batch programs can be found in the test section. at the location:

/afs/cern.ch/na49/library/local/ROOT/hpux10/root/test/*.cxx

Compiling and Linking with the ROOT libraries

Below are the recommended compile and link sequences (for up to date link options for your platform check the `Makefile` in the `root/test` directory).

HP-UX

to compile: `CC -w -O +a1 -I$ROOTSYS/include -c` *`file`*`.C` to link: `CC -w -O +a1 -Wl,-E -o` *`file file`*`.o $ROOTSYS/lib/*.sl -lXm -lXt -lX11 -lm -lPW -ldld` **Note** that the option **`+a1`** is essential when compiling with CC.

AIX 4

to compile: `xlC -w -O -I$ROOTSYS/include -I/usr/lpp/xlC/include -I/usr/include` *`file`*`.C` to link: `xlC -w -O -o` *`file file`*`.o -L$ROOTSYS/lib -lRoot -lCint -lXm -lXt -lX11 -lm -lPW`

Solaris

to compile: `CC -w -O -I$ROOTSYS/include -c` *`file`*`.C` to link: `CC -w -O -o` *`file file`*`.o $ROOTSYS/lib/*.so -L/usr/dt/lib -L/usr/openwin/lib -L/usr/ccs/lib -lXm -lXt -lX11 -lm -lgen`

Version 1.00

Availability

Via Anonymous FTP

Version 1.00 of the ROOT Development Kit (RDK) is available via anonymous ftp for the following platforms:

- Complete source code (version 1.00/10)(for all systems). However, no `Makefile` yet. We hope to include final sources with `autoconf` functionality in the coming weeks. Unpack tar file in your `root` directory where it will create the `src` directory.
- Linux (3.7 MB). You need to have a Motif run-time environment installed to be able to use ROOT. The latest version (>=0.77) of LessTif, a public domain clone of OSF/Motif 1.2, works (after installing LessTif's `libXm.so.0` create a link `libXm.so.2` to it).
- HP-UX 9.x (5 MB). In case you are running HP-UX 9.x and don't have the HP C++ compiler installed you also should download the C++ runtime library libC.tar.gz (61 KB). Unzip and untar before usage.
- HP-UX 10.20 with CC compiler (5 MB).
- HP-UX 10.20 with aCC compiler (5 MB).
- DEC/Alpha/Unix with DEC/CXX (4.3 MB). This library has been compiled under Alpha/Unix version 4.0. To use this library, you must have DEC C++ V5.5-004. Older versions of CXX cannot be used.
- DEC/Alpha/Unix with g++ (5.2 MB).
- AIX 4.1 (4.3 MB, works only on AIX 4.4).
- AIX 3.2 (4.1 MB, works on AIX3.2) Still version 0.9.
- Sun/Solaris for Sun hardware (3.6 MB). Binaries will work OK under Solaris5.4 or beyond. Note that with previous versions of Solaris, the Motif libraries were not part of the system.
- Sun/Solaris for Intel PCs (3 MB). These binaries have been compiled under Solaris5.5.1.
- SGI (4.8 MB). These binaries have been compiled under IRIX5.3 (there are still some X/Motif problems when opening multiple windows, we are working on the problem).
- Windows/NT: all libraries and executables (4.0 MB) self extracting executable modules only (2.0 MB). Use Winzip to unpack and run setup.bat. The executable modules only in 1.4MB portions. Download the 3 files disk1.bin, disk2.bin and setup.bat, then run setup.bat in the directory containing the .bin files. This version has been built using Microsoft **VisualC++**. The X3D facility is missing in the current version under Windows. The **h2root** utility requires a Fortran DLL called MSFRT40.DLL. In case you do not have this library already installed on your system, you can ftp it.
- Windows/95: all libraries and executables (4.0 MB) self extracting executable modules only (2.0 MB). Use Winzip to unpack and run setup.bat. The executable modules only in 1.4MB portions. Download the 3 files disk1.bin, disk2.bin and setup.bat, then run setup.bat in the directory containing the .bin files. This version has been built using Microsoft **VisualC++**. If you don't have yet OpenGL installed take the file OGLFIX.EXE. The X3D facility is missing in the current version under Windows. The **h2root** utility requires a Fortran DLL called MSFRT40.DLL. In case you do not have this library already installed on your system, you can ftp it.
- MAC MkLinux (3 MB). The MkLinux (Linux for Mac) version misses h2root (will be available in a week or so). You must have shared library support installed, the shared library version of X11R6.3, as well as Lesstif 0.77 libraries (shared version) This version of ROOT was only tested on MkLinux(7600/120) but should work with LinuxPMac. If you have problems on installation, please contact D. Buskulic buskulic@lapp.in2p3.fr . If a monolithic (non shared) version is needed, we will try to send it to you. No firm promises at this point, since the first attempt to port ROOT without shared libraries failed.

To use the system, unzip and untar the file. For example:

```
$ gunzip root_v1.00.HP-UX.B.10.20.tar.gz
```

$ tar xvf root_v1.00.HP-UX.B.10.20.tar

This will create the directory `root`. Before getting started read the file `root/AA_README`. *Remember*, you can find the full documentation of the ROOT classes on this web site at the location Classes and Members. The RDK also contains all tutorials and a set of test programs.

The ROOT Team

The ROOT project was started by **René Brun** late November 1994. His long time collaborator **Fons Rademakers** joined the project around January 1995. In the middle of August 1995 **Nenad Buncic** joined the team, followed by **Valery Fine** in December 1995.

René Brun is working at CERN since 1973. During this time he has been instrumental in the design and development of many popular software packages, a.o., HBOOK, GEANT, ZEBRA, PAW and PIAF. Quite a number of these packages found their way out of the High Energy Physics community and are being used by people in many different research areas. For example, during its prime PAW was, according to an estimate, being used by around 20000 people world-wide. Currently René is working in the NA49 collaboration, which provides the stimulating and challenging environment necessary for the development of an ambitious project like ROOT.

Fons Rademakers has been working with René Brun since 1988. Together they worked on CMZ, HBOOK, PAW, PAW++ and PIAF. After obtaining his Ph.D. in Particle Physics Fons has been a CERN fellow for 3 years, and an HP Labs fellow for 1 year. Currently he is working for HP at CERN.

Nenad Buncic is fairly new in the world of High Energy Physics. He has a degree in computer science and is being paid by the NA49 collaboration to work on ROOT. In very short time he managed to grasp the concepts of the system and contributed the 3D graphics, object browser and HTML documentation generation sub-systems.

Valery Fine has a Ph.D in Physics and Mathematics and is a professor of the Dubna University and senior research worker of the Laboratory of Computer Techniques and Automation of the Joint Institute for Nuclear Research (JINR). Valery has ported the ROOT system on the Windows/NT and Windows95 platforms. This work is also supported by NA49.

The authors would like to thank and acknowledge the following people

Masaharu Goto for the CINT C/C++ interpreter

Gunter Roland very early adopter and ROOT evangelist

Jacek Holeczek for the invaluable help in porting and testing ROOT on AIX

Maarten Ballintijn for porting ROOT to Linux on a snowless sunday afternoon

Bjarne Stroustrup for discussions and comments concerning the class structure

P. Bemis and M. Benard Hewlett-Packard Company **R. Stock** NA49 Spokesman **A. Sandoval** NA49 Computer Infrastructure Coordinator **K. Bos** MOOSE Project Coordinator

C. Bormann Early Adopter **M. Marino** Early Adopter

Family and Friends for their support