Preface

In late 1994, we decided to learn and investigate Object Oriented programming and C++ to better judge the suitability of these relatively new techniques for scientific programming. We knew that there is no better way to learn a new programming environment than to use it to write a program that can solve a real problem. After a few weeks, we had our first histogramming package in C++. A few weeks later we had a rewrite of the same package without templates since we could only compile the version with templates on one single platform using the, at that time, very new template features of C++. Again, a few weeks later we had another rewrite of the package without templates since

Since its first public release at the end of 1995, ROOT has enjoyed an ever-increasing popularity. Currently it is being used in all major High Energy and Nuclear Physics laboratories around the world to monitor, to store and to analyze data. In the other sciences as well as in the medical and financial industries, many people are using ROOT. We estimate the current user base to be around several thousand people.

In 1997, Eric Raymond analyzed in his paper "The Cathedral and the Bazaar" the development method that makes Linux such a success. The essence of that method is: "release early, release often and listen to your customers". This is precisely how ROOT is being developed. Over the last five years, many of our "customers" became co-developers. Here we would like to thank our main co-developers and contributors:

Masaharu Goto who wrote the CINT C++ interpreter. CINT has become an essential part of ROOT. Despite being 8 time zones ahead of us, we often have the feeling he is sitting in the room next door.
Valery Fine who ported ROOT to Windows and who also contributed largely to the 3-D graphics and geometry packages.
Nenad Bunic who developed the HTML documentation generation system and integrated the X3D viewer in ROOT.
Philippe Canal who developed the automatic compiler interface to CINT. In addition to a large number of contributions to many different parts of the system, Philippe is also the ROOT support coordinator at FNAL.
Suzanne Panacek who is the main author of this manual. Suzanne is also very active in preparing tutorials and giving lectures about ROOT.

Further, we would like to thank the following people for their many contributions, bug fixes, bug reports and comments:

Maarten Ballintijn, Stephen Bailey, Damir Buskucilic, Federico Carminati, Mat Dobbs, Rutger v.d. Eijk, Anton Fokin, Nick van Eijndhoven, George Heintzelman, Marc Hemberger, Christian Holm Cristensen, Jacek M. Holeczek, Stephan Kluth, Marcel Kunze, Christian Lacunza, Matthew D. Langston, Michal Lijowski, Peter Malzacher, Dave Morrison, Eddy Offermann, Pasha Murat, Valeriy Onuchin, Victor Perevozchikov, Sven Ravndal, Reiner Rohls, Gunther Roland, Andy Salnikov, Otto Schaele, Alexandre V. Vanachine, Torre Wenaus and Hans Wenzel, and many more who have also contributed

You all helped in making ROOT a great experience.

Happy ROOTing!

Rene Brun & Fons Rademakers
# Table of Contents

## Preface

- Table of Contents

## Introduction

1. The ROOT Mailing List
2. Conventions Used in This Book
3. What is a Framework?
4. Why Object-Oriented?
5. The Organization of the ROOT Framework
6. $ROOTSYS/bin
7. $ROOTSYS/Sib
8. $ROOTSYS/tutorials
9. $ROOTSYS/test
10. $ROOTSYS/include
11. How to Find More Information

## Getting Started

13. Start and Quit a ROOT Session
15. Exit ROOT
19. The ROOT Command Line
20. CINT Extensions
20. Helpful Hints for Command Line Typing
21. Multi-line Commands
21. Conventions
21. Coding Conventions
22. Machine Independent Types
22. TObject
23. Global Variables
23. gROOT
23. gFile
23. gDirectory
24. gPad
24. gRandom
24. gEnv
24. History File
25. Environment Setup

## Histories

- History File
- gEnv
- gRandom

## Example of a TProfile2D histogram

## Graphs

- TGraph
- gEnv
- gRandom

## Example of a TProfile

- The TProfile Constructor

## Conventions

- Multi-line Commands

## Helpful Hints for Command Line Typing

- Setting Tick Marks on the Axis
- Giving Titles to the X, Y and Z Axis
- The SCATTER Plot Option
- The ARROW Option
- The BOX Option
- The ERRO Bar Options
- The COLOR Option
- The TEXT Option
- The CONTour Options
- The LEGO Options
- The SURFACE Options
- The BAR options
- Horizontal BAR chart

## How to Find More Information

- $ROOTSYS/<library>
- $ROOTSYS/include
- $ROOTSYS/test
- $ROOTSYS/lib
- The ROOT Command Line

## Memory Checking

- Memory Checker
- Tracking Memory Leaks

## The Organization of the ROOT Framework

- Installing ROOT
- Why Object-Oriented?

## Setting Tick Marks on the Axis

- Setting Tick Marks on the Axis
- Giving Titles to the X, Y and Z Axis
- The SCATTER Plot Option
- The ARROW Option
- The BOX Option
- The ERRO Bar Options
- The COLOR Option
- The TEXT Option
- The CONTour Options
- The LEGO Options
- The SURFACE Options
- The BAR options
- Horizontal BAR chart

## How to Find More Information

- $ROOTSYS/<library>
- $ROOTSYS/include
- $ROOTSYS/test
- $ROOTSYS/lib
- The ROOT Command Line

## Memory Checking

- Memory Checker
- Tracking Memory Leaks

## The Organization of the ROOT Framework

- Installing ROOT
- Why Object-Oriented?

## Setting Tick Marks on the Axis

- Setting Tick Marks on the Axis
- Giving Titles to the X, Y and Z Axis
- The SCATTER Plot Option
- The ARROW Option
- The BOX Option
- The ERRO Bar Options
- The COLOR Option
- The TEXT Option
- The CONTour Options
- The LEGO Options
- The SURFACE Options
- The BAR options
- Horizontal BAR chart

## How to Find More Information

- $ROOTSYS/<library>
- $ROOTSYS/include
- $ROOTSYS/test
- $ROOTSYS/lib
- The ROOT Command Line

## Memory Checking

- Memory Checker
- Tracking Memory Leaks
10 Folders And Tasks
Folders ........................................................................................................ 171
Why Use Folders? ................................................................................... 171
How to Use Folders .............................................................................. 171
Creating a Folder Hierarchy .............................................................. 172
Posting Data to a Folder (Producer) ................................................ 173
Reading Data from a Folder (Consumer) .............................................. 173
Tasks: ...................................................................................................... 174
Execute and Debug Tasks ..................................................................... 177

11 Input/Output
The Physical Layout of ROOT Files .................................................. 179
The File Header ..................................................................................... 181
The Top Directory Description .......................................................... 181
The Class Description List (StreamerInfo list) ......................... 182
The List of Keys and The List of Free Blocks .............................. 184
File Recovery ......................................................................................... 184
The Logical ROOT File: TFile and TKey ....................................... 184
The Current Directory .......................................................................... 188
Objects in Memory and Objects on Disk ......................................... 189
Saving Histograms to Disk ................................................................. 191
Histograms and the Current Directory ........................................... 193
Saving Objects to Disk ......................................................................... 194
Saving Collections to Disk ................................................................. 194
A TFile Object going Out of Scope .................................................... 194
Retrieving Objects from Disk ............................................................. 195
Subdirectories and Navigation .......................................................... 199
Streamers .............................................................................................. 196
Streamer Pointers ................................................................................. 198
Automatically Generated Streamers .............................................. 199
Transient Data Members (///) ............................................................ 200
The Pointer To Objects (//->) ............................................................ 200
Variable Length Array ....................................................................... 201
Prevent Splitting (//) ........................................................................... 201
Streamers With Special Additions ....................................................... 201
Writing Objects ..................................................................................... 202
Ignore Object Streamers ...................................................................... 203
Streaming a TClonesArray ................................................................. 203
Pointers and References in Persistence ............................................. 204
Streaming Pointers .............................................................................. 204
Motivation for the TRef Class ............................................................. 204
Using TRef.......................................................................................... 205
How does it work? .............................................................................. 205
Action on Demand .............................................................................. 207
Array of TRef ....................................................................................... 208
Schema Evolution ................................................................................. 209
The StreamerInfo Class ...................................................................... 210
Example: TH1 StreamerInfo ............................................................... 211
The StreamerInfoElement Class ....................................................... 211
Optimized StreamerInfo ...................................................................... 212
Automatic Schema Evolution ............................................................. 212
Manual Schema Evolution ................................................................. 213
Building Class Definitions With The StreamerInfo ...................... 213
Example: MakeProject ....................................................................... 213
Migrating to ROOT 3 .............................................................................. 217
Compression and Performance ........................................................ 218
Accessing ROOT Files Remotely via a rooTd ...................................... 219
TNetFile URL ....................................................................................... 219

12 Trees
Why should you Use a Tree? ............................................................... 223
A Simple Tree ........................................................................................ 223
Show an Entry with TTree::Show ....................................................... 225
Print the tree structure with TTree::Print ......................................... 225
Set a Variable to NULL with TTree::SetNull .................................... 225
The TTree Viewer ............................................................................... 226
Creating and Saving Trees .................................................................. 226
Creating a Tree from a Folder Hierachy ........................................... 226
Accessing Branches ............................................................................ 226
Benchmarks ......................................................................................... 226
Adding a Branch to hold a List of Variables ...................................... 226
Adding a Branch to hold an Object .................................................... 226
Setting the Split-level ......................................................................... 226
Exempt a Data Member from Splitting ............................................. 226
Adding a Branch to hold a Class and a Member Class .................. 226
Adding a Branch with TBranchContent ......................................... 226
Adding a Branch with TBranchList .................................................. 226
Adding a Branch with TBranchCollection ..................................... 226
Examples For Writing and Reading Trees ....................................... 226
Example 3: A Tree with Simple Variables ....................................... 227
Writing the Tree ................................................................................... 227
Viewing the Tree ............................................................................... 227
Roots ................................................................................................. 228
Example 1: A Tree with Simple Variables ....................................... 228
Writing the Tree ................................................................................... 228
Viewing the Tree ............................................................................... 228
Roots ................................................................................................. 228
Example 2: A Tree with a Structure ................................................. 228
Writing the Tree ................................................................................... 229
Example 3: Adding Friends to Trees .................................................. 229
Adding a TBranch to hold a Class .................................................... 229
Example 4: A Tree within an Event Class ........................................... 229
The Event Class .................................................................................... 230
The Event Header Class ....................................................................... 230
The Track Class .................................................................................... 230
Writing the Tree ................................................................................... 230
Reading the Tree ............................................................................... 230
Trees in Analysis .................................................................................. 230
Simple Analysis using TTree::Draw ................................................ 230
Using Selection with TTree::Draw .................................................... 230
Using TCut Objects in TTree::Draw .................................................. 230
Accessing the Histogram in Batch Mode ......................................... 231
Using Draw Options in TTree::Draw ............................................... 231
Superimposing two Histograms ....................................................... 232
Setting the Range in TTree::Draw ..................................................... 232
TTree::Draw Examples ....................................................................... 232
Filling a Histogram ............................................................................... 232
Projecting a Histogram ..................................................................... 232
Using TTree::MakeClass ................................................................... 232
Using TTree::MakeSelector .............................................................. 232
Performance Benchmarks .................................................................. 232
Remote Authentication ....................................................................... 239
A Simple Session ............................................................................... 239
The rooTd Daemon ............................................................................. 239
Starting rooTd via inetd ................................................................. 239
Creating and Accessing folders on other machines .................. 239
Using the General File:Events Function ................................. 239
Reading ROOTFiles via Apsolon Web Server ............................ 240
Using the General:File:Events Function ........................................ 240
20 Automatic HTML Documentation

21 PROOF: Parallel Processing

22 Threads

Threads and Processes
Process Properties
Thread Properties
The Initial Thread
Implementation of Threads in ROOT
Installation
Classes
TThread for Pedestrians
Loading
Creating
Running
TThread in More Detail
Asynchronous Actions
Synchronous Actions: TCondition
Xlib connections
Canceling a TThread
Advanced TThread: Launching a Method in a Thread
Known Problems
Glossary

23 Appendix A: Install and Build ROOT

ROOT Copyright and Licensing Agreement
Installing ROOT
Choosing a Version
Installing Precompiled Binaries
Installing the Source
More Build Options
Setting the Environment Variables
Documentation to Download

24 Index
1 Introduction

In the mid 1990's, René Brun and Fons Rademakers had many years of experience developing interactive tools and simulation packages. They had lead successful projects such as PAW, PIAF and GEANT, and these new technologies had reached their limits. Although still very popular, these tools could not scale up to the challenges offered by the Large Hadron Collider, where the data is a few orders of magnitude larger than anything seen before.

At the same time, computer science had made leaps of progress especially in the area of Object Oriented Design, and René and Fons were ready to take advantage of it.

ROOT was developed in the context of the NA49 experiment at CERN. NA49 has generated an impressive amount of data, around 10 Terabytes per run. This rate provided the ideal environment to develop and test the next generation data analysis.

One cannot mention ROOT without mentioning CINT its C++ interpreter. CINT was created by Masa Goto in Japan. It is an independent product, which ROOT is using for the command line and script processor.

ROOT was, and still is, developed in the "Bazaar style", a term from the book "The Cathedral and the Bazaar" by Eric S. Raymond. It means a liberal, informal development style that heavily leverages the diverse and deep talent of the user community. The result is that physicists developed ROOT for themselves, this made it specific, appropriate, useful, and over time refined and very powerful.

When it comes to storing and mining large amount of data, physics plows the way with its Terabytes, but other fields and industry are ready to use the true and tested technologies physics has invented. In this way, both fields and industries have found ROOT useful and they have started to use it, and have contributed to it.

The development of ROOT is a continuous conversation between users and developers with the line between the two blurring at times and the users becoming co-developers.

In the bazaar view, software is released early and frequently to expose it to thousands of eager co-developers to test, bug report, and contribute possible fixes. More users find more bugs, because more users add different ways of stressing the program. By now, after six years, many, many users have stressed ROOT in many ways, and it is now mature. Most likely, you will find the features you are looking for, and if you have found a hole, you are encouraged to participate in the dialog and post your suggestion or even implementation on roottalk, the ROOT mailing list.

The ROOT Mailing List

You can subscribe to roottalk, the ROOT Mailing list by registering at the ROOT web site: http://root.cern.ch/root/Registration.phtml.

This is a very active list and if you have a question, it is likely that it has been asked, answered, and stored in the archives. Please use the search engine to see if your question has already been answered before sending mail to root talk.

You can browse the roottalk archives at: http://root.cern.ch/maintain/roottalk/AboutRootTalk.html.

You can send your question without subscribing to: roottalk@root.cern.ch.

Contact Information

This book was written by several authors. If you would like to contribute a chapter or add to a section, please contact us. This is the first and early release of this book, and there will be many changes. However, we wanted to follow the ROOT tradition of releasing early and often to get feedback early and catch mistakes. We count on you to send us suggestions on additional topics on the topics that need more documentation. Please send your comments, corrections, suggestions, and questions to rootdoc@root.cern.ch.

Please note that we cannot give you insights into the many capabilities of ROOT. The book begins with the elementary functionality and progresses in complexity reaching the specialized topics at the end.

The experienced user looking for special topics may find these chapters useful: Networking, Writing a Graphical User Interface, Threads, and PROOF: Parallel Processing.

Because this book was written by several authors, you may see some inconsistencies and a "change of voice" from one chapter to the next. We felt we could accept this in order to have the expert explain what they know best.

Conventions Used in This Book

We tried to follow a style convention for the sake of clarity. Here are the few styles we used.

- To show source code in scripts or source files:

```
    { 
      cout << " Hello" << endl; 
      float x = 3.; 
      float y = 5.; 
      int i = 101; 
      cout << x = "hello" y = "hello" i = "hello" << endl; 
    } 
```

- To show the ROOT command line, we show the ROOT prompt without numbers. In the interactive system, the ROOT prompt has a line number (root [12]), for the sake of simplicity we left off the line number.

Bold monotype font indicates text for you to enter at verbatim.

```
    cout << "Hello" << endl; 
    float x = 3.; 
    float y = 5.; 
    int i = 101; 
    cout << x = "hello" y = "hello" i = "hello" << endl; 
```

Italic bold monotype font indicates a global variable, for example gDirectory.

```c
    root[] TLine 1 
    root[] l.Print() 
    TLine X1=0.000000 Y1=0.000000 X2=0.000000 Y2=0.000000 
```

Italic bold monotype font indicates a global variable, for example gDirectory.
We also used the italic bold font to highlight the comments in the code listing.

When a variable term is used, it is shown between angled brackets. In the $ROOTSYS directory, $ROOTSYS/<library>/inc

The Framework

ROOT is an object-oriented framework aimed at solving the data analysis challenges of high-energy physics. There are two key words in this definition: object oriented and framework. First, we explain what we mean by a framework and then why it is an object-oriented framework.

What is a Framework?

Programming inside a framework is a little like living in a city. Plumbing, electricity, telephone, and transportation are services provided by the city. In your house, you have interfaces to the services such as light switches, electrical outlets, and telephones. The details, for example the routing algorithm of the phone switching system, are transparent to you as the user. You do not care, you are only interested in using the phone to communicate with your collaboration to solve your domain specific problems.

Programming outside of a framework may be compared to living in the country. In order to have transportation and water, you will have to build a road and dig a well. The details of building the infrastructure (or framework) before you can use the phone to communicate

In software engineering, it is much the same way. In a framework the basic utilities and services, such as I/O and graphics, are provided. In addition, ROOT being a HEP analysis framework, it provides a large selection of HEP specific utilities such as histograms, trees. The routing algorithm of the framework is that you are constrained to it, as you are constrained to use the framework that provide your telephone service. You also have to learn the framework interfaces, which is much the same as learning how to use a telephone.

If you are interested in doing physics, a good HEP framework will save you much work.

Below is a list of the more commonly used components of ROOT:

- Command Line Interpreter
- Histograms and Fitting
- Graphic User Interface widgets
- 2D Graphics
- I/O
- Collection Classes
- Script Processor

There are also less commonly used components, these are:

- 3D Graphics

Advantages of Frameworks

The benefits of frameworks can be summarized as follows:

- Less code to write: The programmer should be able to use and reuse the majority of the code. Basic functionality, such as fitting and histogramming are implemented and ready to use and customize.
- More reliable and robust code: Code inherited from a framework has already been tested and integrated with the rest of the framework.
- More consistent and modular code: Code reuse provides consistency in common operations between programs, in contrast with separate programs.
- More focus on areas of expertise: Users can concentrate on their particular problem domain. They don’t have to be experts at writing user interfaces, graphics, or networking to use the frameworks that provide these services.

Why Object-Oriented?

Object-Oriented Programming offers considerable benefits compared to Procedure-Oriented Programming:

- Encapsulation: it enforces data abstraction and increases opportunity for reuse.
- Subclassing and inheritance make it possible to extend and modify existing code.
- Class hierarchies and containment hierarchies provide a flexible mechanism for modeling real-world objects and the relationships among them.
- Complexity is reduced because there is little growth of the global state, the state is contained within each object, rather than scattered through the program in the form of global variables.
- Objects may come and go, but the basic structure of the program remains relatively static, increases opportunity for reuse of design.

Installing ROOT

The installation and building of ROOT is described in Appendix A: Install and Build ROOT. You can download the binaries (7 MB to 11 MB depending on the platform), or the source (about 3.4 MB). ROOT can be compiled by the GNU g++ compiler on most Unix platforms.

ROOT is currently running on the following platforms:

- Intel x86 Linux (g++, egcs and KAI/KCC)
- Intel Itanium Linux (g++)
- HP Itanium Linux (HP CC and aCC, egcs1.2 C++ compilers)
- IBM AIX 4.1 (xlC compiler and egcs1.2)
- Sun Solaris for SPARC (SUN C++ compiler and egcs)
- Sun Solaris for x86 (SUN C++ compiler)
- Sun Solaris for x86 KAI/KCC
- Compaq Alpha OSF1 (egcs1.2 and DEC/CXX)
The Organization of the ROOT Framework

Now we know in abstract terms what the ROOT framework is, let's look at the physical directories and files that come with the installation of ROOT.

You may work on a platform where your system administrator has already installed ROOT. You will need to follow the specific development environment for your setup and you may not have write access to the directories. In any case, you will need an environment variable called $ROOTSYS, which holds the path of the top directory.

```
$ echo $ROOTSYS
/home/root
```

In the $ROOTSYS directory are examples, executables, tutorials, header files, and if you opted to download the source it is also here. The directories of special interest to us are bin, tutorials, lib, test, and include. The diagram on the next page shows the contents of these directories.
$ROOTSYS/bin

The bin directory contains several executables:

- root: shows the ROOT splash screen and calls root.exe
- rootd: is the daemon for remote file access. It is run in the background or directly
- proofd: is the daemon for remote process access
- makecint: is the utility ROOT uses to create a class dictionary for CINT
- rootcint: is the pure CINT version of C++ interpreter
- cint: is the interactive interface to ROOT (provides command prompt)
- gdb: is the Thread classes
- proofserv: is a small daemon used to authenticate a user of ROOT, has successfully been authenticated.
- proofd: is the actual PROOF process, which is started byproofserv

$ROOTSYS/lib

There are several ways to use ROOT, one way is to run the executable by typing root at the system prompt. Another way is to link with the ROOT libraries and make the ROOT classes available in your own program.

Here is a short description for each library, the ones marked with a * are only installed when the options specified them:

- libCint: is the C++ interpreter (CINT)
- libCore: is the Base classes
- libEG: is the abstract event generator interface classes
- libEGPythia5.so: is the Pythia5 event generator interface
- libEGPythia6.so: is the Pythia6 event generator interface
- libEGVenus.so: is the Venus event generator interface
- libGpad: is the pad and canvas classes which depend on low level graphics
- libGx11: is the 2D graphics primitives (can be used independent of libGpad.so)
- libGx11TTF: is the TrueType fonts
- libHist: is the histogram classes
- libHistPainter: is the histogram painting classes
- libHtml: is the HTML documentation generation system
- libMatrix: is the matrix and vector manipulation
- libMinuit: is the MINUIT fitter
- libNew: is the special global new/delete, provides extra memory checking and interface for shared memory (optional)
- libPhysics: is the physics quantity manipulation classes (TLorentzVector, etc.)
- libPostScript: is the PostScript interface
- libProof: is the parallel ROOT Facility classes
- *libRFIO: is the interface to CERN RFIO remote I/O system
- *libRGL: is the interface to OpenGL
- *libThread: is the interactive interface to ROOT (provides command prompt)
- *libTree: is the Tree classes
- *libTreePlayer: is the Tree object container system
- *libTreePlayer.so: is the Tree drawing classes
- *libTreeViewer: is the graphical Tree query interface
- *libX3d: is the X3D system used for fast 3D display

Library Dependencies

The libraries are designed and organized to minimize dependencies, such that you can include just enough code for the task at hand rather than having to include all libraries or one monolithic chunk.

The core library (libCore.so) contains the essentials it needs to be included in all ROOT applications. In the diagram, you see that libCore is made up of Base classes, Container classes, Meta information classes, Networking classes, Operating system specific classes and the ZIP algorithm used for compression of the ROOT files.

The CINT library (libCint.so) is also needed in all ROOT applications, but CINT can be used independently of libCore, if you only need the C++ interpreter and no ROOT. That is the reason these two are separate.

A program referencing only libCore only need libCint and libCore.

This includes the ability to read and write ROOT objects, and there are no dependencies on graphics or the GUI.

![Diagram of ROOT libraries and dependencies](image-url)
A batch program, one that does not have a graphic display, which creates, fills, and saves histograms and trees, only needs the core (libCore and libCint), libHist and libTree. If other libraries are needed, ROOT loads them dynamically. For example if the TreeViewer is used, libTreePlayer and all the libraries the TreePlayer box below has an arrow to will be loaded automatically at runtime when needed. All the libraries outside of the core are automatic. Of course, if one wants to access an automatic library directly, it has to be explicitly linked also.

In this case:

- GPad
- Graf3d
- Graf
- HistPainter
- Hist
- Tree

The difference between libHist and libHistPainter is that the former needs to be explicitly linked and the latter will be loaded automatically at runtime when needed. In the diagram, the dark boxes outside of the core are automatic. Of course, if one wants to access an automatic library directly, it has to be explicitly linked also.

An example of a dynamically linked library is Minuit. To create and fill histograms you need to link libHist. If the code has a call to fit the histogram, the "Fitter" will check if Minuit is already loaded and if not it will dynamically load it.

The tutorials directory contains many example scripts. They assume some basic knowledge of ROOT, and for the new user we recommend reading the chapters: Histograms and Input/Output before the chapter Tutorials and Tests to find more explicit and specific information about how to build and run the examples.

The $ROOTSYS/test directory is a gold mine of ROOT-wisdom nuggets, and we encourage you to explore and exploit it. However, we recommend that the new user read the chapters: The chapter Tutorials and Tests, has instructions on how to build all the programs and goes over the examples Event and stress.

The $ROOTSYS/tutorials directory contains many example scripts. They assume some basic knowledge of ROOT and for the new user we recommend reading the chapters: Histograms and Input/Output before the examples. The more experienced user will want to look at the Tutorial and Tests to find more explicit and specific information about how to build and run the examples.

The $ROOTSYS/tutorials directory contains many example scripts. They assume some basic knowledge of ROOT and for the new user we recommend reading the chapters: Histograms and Input/Output before the examples. The more experienced user will want to look at the Tutorial and Tests to find more explicit and specific information about how to build and run the examples.

The $ROOTSYS/tutorials directory contains many example scripts. They assume some basic knowledge of ROOT and for the new user we recommend reading the chapters: Histograms and Input/Output before the examples. The more experienced user will want to look at the Tutorial and Tests to find more explicit and specific information about how to build and run the examples.
How to Find More Information

The ROOT website has up to date documentation. The ROOT source code automatically generates this documentation, as each class is explicitly documented on its own web page, which is always up to date with the latest official release of ROOT. The class index web pages can be found at the root page [http://root.cern.ch/root/html/ClassIndex.html](http://root.cern.ch/root/html/ClassIndex.html). Each page contains a class description, and an explanation of each method. It shows the class it was derived from and lets you jump to the parent class page by clicking on the class name. If you want more details, you can examine the source. In addition to this, the site contains tutorials, “How To’s”, and a list of publications and example applications.
2 Getting Started

We begin by showing you how to use ROOT interactively. There are two examples to click through and learn how to use the GUI. We continue by using the command line, and explaining the coding conventions, global variables and the environment setup.

If you have not installed ROOT, you can do so by following the instructions in the appendix, or on the ROOT web site: http://root.cern.ch/root/Availability.html

Start and Quit a ROOT Session

To start ROOT you can type_root_at_the_system_prompt_. This starts up CINT the ROOT command line C/C++ interpreter, and it gives you the ROOT prompt (_root_ [0]).

\%
root
***************************************************
*                                                *
*        W E L C O M E  to  R O O T            *
*                                                *
*   Version   2.25/02      21 August 2000       *
*                                                *
*  You are welcome to visit our Web site         *
*          http://root.cern.ch                *
*                                                *
***************************************************
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.

\%
\%

It is possible to launch ROOT with some command line options, as shown below:

% root -b -q
Usage: root [-l] [-b] [-n] [-q] [file1.C ... fileN.C]
Options:
-b : run in batch mode without graphics
-n : do not execute logon and logoff macros as
      specified in .rootrc
-q : exit after processing command line script files
-l : do not show the image logo (splash screen)

--b: Run in batch mode, without graphics display. This mode is useful in case
one does not want to set the DISPLAY or cannot do it for some reason.
--n: Usually, launching a ROOT session will execute a logon script and
quitting will execute a logoff script. This option prevents the execution of
these two scripts.

It is also possible to execute a script without entering a ROOT session. One
simply adds the name of the script(s) after the ROOT command. Be warned:
after finishing the execution of the script, ROOT will normally enter a new
session.
--q: exit after processing command line script files. Retrieving previous
commands and navigating on the Command Line.

For example if you would like to run a script in the background, exit after
execution, and redirect the output into a file, use the following syntax:

root -b -q myMacro.C > myMacro.log

For a quicker execution (i.e. compiled speed rather than interpreted speed),
you can build a shared library with ACLiC (see the Chapter on CINT) and
then use the shared library on the command line.

root -b -q myMacro.so > myMacro.log

ROOT's powerful C/C++ interpreter gives you access to all available ROOT
classes, global variables, and functions via a command line. By typing C++
statements at the prompt, you can create objects, call functions, execute
scripts, etc. For example:

root[] list=gt(9);
(double)4.0000000000000e+00
root[]for (int i = 0; i<5; i++) cout << "Hello": << i << endl
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
root[] .q

It is possible to launch ROOT with some command line options, as shown
below:

% root -b -q
Usage: root [-l] [-b] [-n] [-q] [file1.C ... fileN.C]
Options:
-b : run in batch mode without graphics
-n : do not execute logon and logoff macros as
      specified in .rootrc
-q : exit after processing command line script files
-l : do not show the image logo (splash screen)

--b: Run in batch mode, without graphics display. This mode is useful in case
one does not want to set the DISPLAY or cannot do it for some reason.
--n: Usually, launching a ROOT session will execute a logon script and
quitting will execute a logoff script. This option prevents the execution of
these two scripts.

It is also possible to execute a script without entering a ROOT session. One
simply adds the name of the script(s) after the ROOT command. Be warned:
after finishing the execution of the script, ROOT will normally enter a new
session.
--q: exit after processing command line script files. Retrieving previous
commands and navigating on the Command Line.

For example if you would like to run a script in the background, exit after
execution, and redirect the output into a file, use the following syntax:

root -b -q myMacro.C > myMacro.log

For a quicker execution (i.e. compiled speed rather than interpreted speed),
you can build a shared library with ACLiC (see the Chapter on CINT) and
then use the shared library on the command line.

root -b -q myMacro.so > myMacro.log

ROOT's powerful C/C++ interpreter gives you access to all available ROOT
classes, global variables, and functions via a command line. By typing C++
statements at the prompt, you can create objects, call functions, execute
scripts, etc. For example:

root[] list=gt(9);
(double)4.0000000000000e+00
root[]for (int i = 0; i<5; i++) cout << "Hello": << i << endl
Hello 0
Hello 1
Hello 2
Hello 3
Hello 4
root[] .q
Exit ROOT
To quit the command line type .q.
TCanvas. Try clicking on these and observe the context menu with their methods.

For the function, try for example to select the `SetRange` method and put -10, 10 in the dialog box fields. This is equivalent to executing the member function `f1.SetRange(-10,10)` from the command line prompt, followed by `f1.Draw()`.

Here are some other options you can try. For example, select the Draw Panel item of the popup menu.

You will see a panel like this:

Once the picture suits your wishes, you may want to see the code you should put in a script to obtain the same result. To do that, choose the "Save as canvas.C" option in the "File" menu. This will generate a script showing the various options. Notice that you can also save the picture in PostScript or GIF format.

One other interesting possibility is to save your canvas in native ROOT format. This will enable you to open it again and to change whatever you like, since all the objects associated to the canvas (histograms, graphs) are saved at the same time.
Second Example: Building a Multi-pad Canvas

Let’s now try to build a canvas (i.e. a window) with several pads. The pads are sub-windows that can contain other pads or graphical objects.

```cpp
root[] TCanvas *MyC = new TCanvas("MyC","Test canvas",1)
root[] MyC->Divide(2,2)
```

Once again, we called the constructor of a class, this time the class TCanvas. The difference with the previous constructor call is that we want to build an object with a pointer to it.

Next, we call the method Divide of the TCanvas class (that is TCanvas::Divide()), which divides the canvas into four zones and sets up a pad in each of them.

```cpp
root[] MyC->cd(1)
root[] f1->Draw()
```

Now, the function f1 will be drawn in the first pad. All objects will now be drawn in that pad. To change the active pad, there are three ways:

- Click on the middle button of the mouse on an object, for example a pad. This sets this pad as the active one
- Use the method TCanvas::cd with the pad number, as was done in the example above:

```cpp
root[] MyC->cd(3)
```

Pads are numbered from left to right and from top to bottom.

Each new pad created by TCanvas::Divide has a name, which is the name of the canvas followed by _1, _2, etc. For example to apply the method cd() to the third pad, you would write:

```cpp
root[] MyC_3->cd()
```

This lists the different constructors and parameters for TLine.

The third pad will be selected since you called TPad::cd() for the object MyC_3. ROOT automatically found the pad that was named MyC_3 when you typed it on the command line (see ROOT/CINT Extensions to C++).

The obvious question is: what is the relation between a canvas and a pad? In fact, a canvas is a pad that spans through an entire window. This is nothing else than the notion of inheritance. The TPad class is the parent of the TCanvas class.

Printing the Canvas

To print a canvas click on the File menu and select Print. This will create a postscript file containing the canvas. The file is named <canvasname>.ps. Then you can send the postscript file to your printer.

The ROOT Command Line

We have briefly touched on how to use the command line, and you probably saw that there are different types of commands.

1. CINT commands start with “.”

```cpp
root[] .?
// this command will list all the CINT commands
root[] .L <filename>
// load [filename]
root[] .X <filename>
// load [filename] and execute function [filename]
```

2. SHELL commands start with “!”, for example:

```cpp
root[] .! ls
```

3. C++ commands follow C++ syntax (almost)

```cpp
root[] TBrowser *b = new TBrowser()
```

CINT Extensions

We can see that some things are not standard C++. The CINT interpreter has several extensions. See the section ROOT/CINT Extensions to C++ in chapter CINT the C++ Interpreter.

Helpful Hints for Command Line Typing

The interpreter knows all the classes, functions, variables, and user defined types. This enables ROOT to help the user complete the command line. For example we do not know yet anything about the TLine class. We can use the Tab feature to get help. Where <TAB> means type the <TAB> key. This lists all the classes starting with TL.

```cpp
root[] l = new TL<TAB>
TLleaf
TLleafB
TLleafC
TLleafD
TLleafF
TLleafI
TLleafObject
TLleafS
TLLine
TLlatex
TLegendEntry
TLegend
TLink
TList
TListIter
TLazyMatrix
TLazyMatrixD
```

This lists the different constructors and parameters for TLine.
Multi-line Commands

You can use the command line to execute multi-line commands. To begin a multi-line command you must type a single left curly bracket {, and to end it you must type a single right curly bracket }.

For example:

```cpp
root[] { 
    for (Int_t i = 0; i < 3; i++) {
        j = j + i;
        cout << "i = " << i <<", j = " << j <<endl;
    }
}
```

It is more convenient to edit scripts than the command line, and if your multi-line commands are getting unmanageable you may want to start a script instead.

Conventions

In this paragraph, we will explain some of the conventions used in ROOT source and examples.

Coding Conventions

From the first days of ROOT development, it was decided to use a set of coding conventions. This allows a consistency throughout the source code. Learning these will help you identify what type of information you are dealing with and enable you to understand the code better and quicker. Of course, you can use whatever convention you want but if you are going to submit some code for inclusion into the ROOT sources you will need to use these. These are the coding conventions:

- Classes begin with T:
  ```cpp
  TTree, TBranch
  ```
- Non-class types end with _t:
  ```cpp
  Int_t, UChar_t
  ```
- Data members begin with f:
  ```cpp
  fTree
  ```
- Member functions begin with a capital:
  ```cpp
  Loop()
  ```
- Constants begin with k:
  ```cpp
  kInitialSize, kKed
  ```
- Global variables begin with g:
  ```cpp
  gEnv
  ```
- Static data members begin with fg:
  ```cpp
  fgTokenClient
  ```
- Enumeration types begin with E:
  ```cpp
  EColorLevel
  ```
- Locals and parameters begin with a lower case:
  ```cpp
  nbytes
  ```
- Getters and setters begin with Get and Set:
  ```cpp
  SetLast(), GetFirst()
  ```

Machine Independent Types

Different machines may have different lengths for the same type. The most famous example is the int type. It may be 16 bits on some old machines and 32 bits on some newer ones.

To ensure the size of your variables, use these predefined types in ROOT:

- Char_t: Signed Character 1 byte
- Uchar_t: Unsigned Character 1 byte
- Short_t: Signed Short integer 2 bytes
- UShort_t: Unsigned Short integer 2 bytes
- Int_t: Signed integer 4 bytes
- UInt_t: Unsigned integer 4 bytes
- Long_t: Signed long integer 8 bytes
- ULong_t: Unsigned long integer 8 bytes
- Float_t: Float 4 bytes
- Double_t: Float 8 bytes
- Bool_t: Boolean (0=false, 1=true)

If you do not want to save a variable on disk, you can use int or Int_t, the result will be the same and the interpreter or the compiler will treat them in exactly the same way.

TObject

In ROOT, almost all classes inherit from a common base class called TObject. This kind of architecture is also used in the Java language. The TObject class provides default behavior and protocol for all objects in the ROOT system. The main advantage of this approach is that it enforces the common behavior of the derived classes and consequently it ensures the consistency of the whole system.

TObject provides protocol, i.e. (abstract) member functions, for:

- Object I/O (Read(), Write())
- Error handling (Warning(), Error(), SysError(), Fatal())
- Sorting (IsSortable(), Compare(), IsEqual(), Hash())
- Inspection (Dump(), Inspect())
- Printing (Print())
- Drawing (Draw(), Paint(), ExecuteEvent())
- Bit handling (SetBit(), TestBit())
- Memory allocation (operator new and delete, IsOnHeap())
- Access to meta information (IsA(), InheritsFrom())
- Object browsing (Browse(), IsFolder())

See "The Role of TObject" in the chapter "Adding a Class".
Global Variables

ROOT has a set of global variables that apply to the session. For example, `gDirectory` always holds the current directory, and `gStyle` holds the current style. All global variables begin with “g” followed by a capital letter.

**gROOT**

The single instance of TROOT is accessible via the global `gROOT` and holds information relative to the current session. By using the `gROOT` pointer you get the access to basically every object created in a ROOT program. The TROOT object has several lists pointing to the main ROOT objects.

**The Collections of gROOT**

During a ROOT session, the `gROOT` keeps a series of collections to manage objects. These can be accessed with the `gROOT::GetListOf` methods.

- `gROOT->GetListOfClasses()`
- `gROOT->GetListOfColors()`
- `gROOT->GetListOfTypes()`
- `gROOT->GetListOfGlobalFunctions()`
- `gROOT->GetListOfFiles()`
- `gROOT->GetListOfMappedFiles()`
- `gROOT->GetListOfCanvases()`
- `gROOT->GetListOfTypes()`
- `gROOT->GetListOfFunction()`
- `gROOT->GetListOfSpecials()`
- `gROOT->GetListOfGeometries()`
- `gROOT->GetListOfBrowsers()`
- `gROOT->GetListOfMessageHandlers()`

These methods return a `TSeqCollection`, meaning a collection of objects, and they can be used to do list operations such as finding an object, or traversing the list and calling a method for each of the members. See the `TCollection` class description for the full set of methods supported for a collection.

For example, to find a canvas called `c1`:

```cpp
gROOT->GetListOfCanvases()->FindObject("c1")
```

This returns a pointer to a `TObject`, and before you can use it as a canvas you will need cast it to a `TCanvas*`.

**gPad**

A graphic object is always drawn on the active pad. It is convenient to access the active pad, no matter what it is. For that we have `gPad` that is always pointing to the active pad. For example, if you want to change the fill color of the active pad to blue, but you do not know its name, you can use `gPad`.

```cpp
gPad->SetFillColor(38)
```

To get the list of colors, if you have an open canvas, click in the “View” menu, selecting the “Colors” entry.

**gRandom**

`gRandom` is a pointer to the current random number generator. By default, it points to a `TRandom` object. Setting the seed to 0 implies that the seed will be generated from the time. Any other value will be used as a constant.

The following basic random distributions are provided:

- `Gaus(mean, sigma)`
- `Rndm()`
- `Landau(mean, sigma)`
- `Poisson(mean)`
- `Binomial(ntot, prob)`

You can customize your ROOT session by replacing the random number generator. You can delete `gRandom` and recreate it with your own:

```cpp
delete gRandom;
gRandom = new TRandom3(0); //seed=0
```

`TRandom3` derives from `TRandom` and is a very fast generator with higher periodicity.

**gEnv**

`gEnv` is the global variable (of type `TEnv`) with all the environment settings for the current session. This variable is set by reading the contents of a `.rootrc` file (or `$ROOTSYS/etc/system.rootrc`) at the beginning of the session. See “Environment Setup” below for more information.

**History File**

You can use the up and down arrow at the command line, to access the previous and next command. The commands are recorded in the history file `$HOME/.root_hist`. It contains the last 100 commands. It is a text file, and you can edit and cut and paste from it.

You can specify the history file in the `system.rootrc` file (see below), by setting the `Rint.History` option. You can also turn off the command logging in the `system.rootrc` file with the option: `Rint.History: -`
Environment Setup

The behavior of a ROOT session can be tailored with the options in the rootrc file. At start-up, ROOT looks for a rootrc file in the following order:

- ./.rootrc //local directory
- $HOME/.rootrc //user directory
- $ROOTSYS/etc/system.rootrc //global ROOT directory

If more than one rootrc file is found in the search paths above, the options are merged, with precedence local, user, global.

While in a session, to see current settings, you can do

```
root[] gEnv->Print()
```

The rootrc file typically looks like:

```
# Path used by dynamic loader to find shared libraries
Unix.*.Root.DynamicPath: .:/rootlibs:$ROOTSYS/lib
Unix.*.Root.MacroPath: .:/rootmacros:$ROOTSYS/macros

# Path where to look for TrueType fonts
Unix.*.Root.UseTTFonts: true
Unix.*.Root.TTFontPath:

# Activate memory statistics
Rint.Root.MemStat: 1
Rint.Load: rootalias.C
Rint.Logon: rootlogon.C
Rint.Logoff: rootlogoff.C

Rint.Canvas.MoveOpaque: false
Rint.Canvas.HighlightColor: 5
```

The various options are explained in $ROOTSYS/etc/system.rootrc.

The .rootrc file contents are combined. For example, if the flag to use true type fonts is set to true in one of the system.rootrc files, you have to explicitly overwrite it and set it to false. Removing the UseTTFonts statement in the local .rootrc file will not disable true fonts.

The Script Path

ROOT looks for scripts in the path specified in the rootrc file in the Root.Macro.Path variable. You can expand this path to hold your own directories.

Logon and Logoff Scripts

The rootlogon.C and rootlogoff.C files are script loaded and executed at start-up and shutdown. The rootlogas.C file is loaded but not executed. It typically contains small utility functions. For example, the rootalias.C script that comes with the ROOT distributions is in the $ROOTSYS/tutorials directory and defines the function edit(char *file). This allows the user to call the editor from the command line. This particular

```
function will start the VI editor if the environment variable EDITOR is not set.
```

```
root [0] edit("cl.C")
```

For more details, see $ROOTSYS/tutorials/rootalias.C.

Tracking Memory Leaks

You can track memory usage and detect leaks by monitoring the number of objects that are created and deleted (see TObjectTable). To use this facility, edit the file .rootrc if you have this file or $ROOTSYS/etc/system.rootrc and edit or add the two following lines:

```
Root.MemStat: 1
Root.ObjectStat: 1
```

In your code, or on the command line you can type the line:

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

This line will print the list of active classes and the number of instances for each class. By comparing consecutive print outs, you can see objects that you forgot to delete.

Note that this method cannot show leaks coming from the allocation of non-objects or classes unknown to ROOT.

Memory Checker

A memory checking system was developed by D.Bertini and M.Ivanov and added in ROOT version 3.02.07. To activate the memory checker you can set the resource Root.MemCheck to 1 (e.g.:

```
Root.MemCheck: 1
```

) in the .rootrc file. You also have to link with libNew.so (e.g. use root-config --new --libs) or use rootn.exe. When these settings are in place, you will find a file "memcheck.out" in the directory where you started your ROOT program after the completion of the program execution.

You can also set the resource Root.MemCheckFile to the name of a file. The memory information will be written to that file. The contents of this memcheck.out can be analyzed and transformed into printable text via the memprobe program (in $ROOTSYS/bin).

Converting HBOOK/PAW files

ROOT has a utility 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 utility converts HBOOK histograms into ROOT histograms 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 the chapter on Trees). Some HBOOK column-wise ntuples may not be fully converted if the columns are an array of fixed dimension (e.g. var[6]) or if they are 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 command line. An example of session is shown below:

```c
// this connects the file hbookconverted.root
root[] TFile f("hbookconverted.root");

// display histogram named h10 (was HBOOK id 10)
root[] h10.Draw();

// display column "var" from ntuple h30
root[] h30.Draw("var");
```

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

The chapter on trees explains how to read a Tree. ROOT includes a function TTree::MakeClass to automatically generate the code for a skeleton analysis function (see the chapter Example Analysis).

In case one of the ntuple columns has a variable length (e.g. px[ntrack]), h.Draw("pxn") will histogram the px column for all tracks in the same histogram. Use the script quoted above to generate the skeleton function and create/fill the relevant histogram yourself.
3 histograms

This chapter covers the functionality of the histogram classes. We begin with an overview of the histogram classes and their inheritance relationship. Then we give instructions on the histogram features.

We have put this chapter ahead of the graphics chapter so that you can begin working with histograms as soon as possible. Some of the examples have graphics commands that may look unfamiliar to you. These are covered in the chapter on Input/Output.

The histogram classes

ROOT supports the following histogram types:

1-D histograms:
- TH1C: histograms with one byte per channel. Maximum bin content = 255
- TH1S: histograms with one short per channel. Maximum bin content = 65,535
- TH1F: histograms with one float per channel. Maximum precision 7 digits
- TH1D: histograms with one double per channel. Maximum precision 14 digits

2-D histograms:
- TH2C: histograms with one byte per channel. Maximum bin content = 255
- TH2S: histograms with one short per channel. Maximum bin content = 65,535
- TH2F: histograms with one float per channel. Maximum precision 7 digits
- TH2D: histograms with one double per channel. Maximum precision 14 digits

3-D histograms:
- TH3C: histograms with one byte per channel. Maximum bin content = 255
- TH3S: histograms with one short per channel. Maximum bin content = 65,535
- TH3F: histograms with one float per channel. Maximum precision 7 digits
- TH3D: histograms with one double per channel. Maximum precision 14 digits

Profile histograms:
- TProfile: one dimensional profiles
- TProfile2D: two dimensional profiles

Profile histograms are used to display the mean value of Y and its RMS for each bin in X. Profile histograms are in many cases an elegant replacement of two-dimensional histograms. The inter-relation of two measured quantities X and Y can always be visualized with a two-dimensional histogram or scatter-plot. If Y is an unknown but single-valued approximate function of X, it will have greater precisions in a profile histogram than in a scatter plot.

All histogram classes are derived from the base class TH1. This image shows the class hierarchy of the histogram classes.

The TH*C classes also inherit from the array class TArrayC.
The TH*S classes also inherit from the array class TArrayS.
The TH*F classes also inherit from the array class TArrayF.
The TH*D classes also inherit from the array class TArrayD.

The histogram classes have a rich set of methods. Below is a list of what one can do with the histogram classes.

Creating Histograms

Histograms are created with constructors:

```
TH1F *h1 = new TH1F("h1","h1 title",100,0,4.4);
TH2F *h2 = new TH2F("h2","h2 title",40,0,4,30,-3,3);
```

The parameters to the TH1 constructor are: the name of the histogram, the title, the number of bins, the x minimum, and x maximum.

Histograms may also be created by:
- Calling the Clone method of an existing histogram (see below)
- Making a projection from a 2-D or 3-D histogram (see below)
- Reading a histogram from a file

When a histogram is created, a reference to it is automatically added to the list of in-memory objects for the current file or directory. This default behavior
can be disabled for an individual histogram or for all histograms by setting a
global switch.

Here is the syntax to set the directory of a histogram:

```c
// to set the in-memory directory for h the current histogram
h->SetDirectory(0);
// global switch to disable
TH1::AddDirectory(kFALSE);
```

When the histogram is deleted, the reference to it is removed from the list of
objects in memory. In addition, when a file is closed, all histograms in
memory associated with this file are automatically deleted. See chapter
Input/Output.

### Fixed or Variable Bin Size

All histogram types support fixed or variable bin sizes. 2-D histograms may
have fixed size bins along X and variable size bins along Y or vice-versa. The
functions to fill, manipulate, draw, or access histograms are identical in both
cases.

To create a histogram with variable bin size one can use this constructor:

```c
TH1(const char name,const char* title,Int_t nbins,Float_t *xbins)
```

The parameters to this constructor are:
- `title`: histogram title
- `nbins`: number of bins
- `xbins`: array of low-edges for each bin. This is an array of size `nbins+1`

Each histogram always contains three `TAxis` objects: `fXaxis`, `fYaxis`,
and `fZaxis`. To access the axis parameters first get the axis from the
histogram, and then call the `TAxis` access methods.

```c
TAxis *xaxis = h->GetXaxis();
Double_t binCenter = xaxis->GetBinCenter(bin);
```

See class `TAxis` for a description of all the access methods. The axis range
is always stored internally in double precision.

### Bin numbering convention

For all histogram types: `nbins`, `xlow`, `xup`

Bin# 0 contains the underflow.
Bin# 1 contains the first bin with low-edge (`xlow` INCLUDED).
The second to last bin (bin# `nbins`) contains the upper-edge (`xup` EXCLUDED).
The Last bin (bin# `nbins+1`) contains the overflow.

In case of 2-D or 3-D histograms, a "global bin" number is defined. For
example, assuming a 3-D histogram with `binx`, `biny`, `binz`, the function
returns a global/linear bin number.

```c
Int_t bin = h->GetBin(binx,biny,binz);
```

This global bin is useful to access the bin information independently of the
dimension.

### Re-binning

At any time, a histogram can be re-binned via the `TH1::Rebin` method. It
returns a new histogram with the re-binned contents. If bin errors were
stored, they are recomputed during the re-binning.

### Filling Histograms

A histogram is typically filled with statements like:

```c
h1->Fill(x);
```

The `Fill` method computes the bin number corresponding to the given `x`,
`y` or `z` argument and increments this bin by the given weight. The
`Fill` method returns the bin number for 1-D histograms or global bin number
for 2-D and 3-D histograms. If `TH1::Sumw2` has been called before filling, the
sum of squares is also stored.

One can also increment a bin number directly by calling

```c
TH1::AddBinContent.
```

Replace the existing content via

```c
TH1::SetBinContent, and access the bin content of a given bin via
TH1::GetBinContent.
```

### Automatic Re-binning Option

By default, the number of bins is computed using the range of the axis. You
can change this to automatically re-bin by setting the automatic re-binning
option:

```c
h->SetBit(TH1::kCanRebin);
```

Once this is set, the `Fill` method will automatically extend the axis range to
accommodate the new value specified in the `Fill` argument. The method
used is to double the bin size until the new value fits in the range, merging
bins two by two.

This automatic binning options is extensively used by the `TTree::Draw`
function when drawing histograms of variables in `TTrees` with an unknown
range. The automatic binning option is supported for 1-D, 2-D and 3-D
histograms.

During filling, some statistics parameters are incremented to compute the
mean value and root mean square with the maximum precision. In case of
histograms of type `TH1C`, `TH1S`, `TH2C`, `TH2S`, `TH3C`, `TH3S` a check is
made that the bin contents do not exceed the maximum positive capacity.
Histograms of all types may have positive or negative bin contents.

Random Numbers and Histograms

`TH1::FillRandom` can be used to randomly fill a histogram using the contents of an existing `TH1` function or another `TH1` histogram (for all dimensions). For example, the following two statements create and fill a histogram 10000 times with a default Gaussian distribution of mean 0 and sigma 1:

```cpp
TH1F h1("h1","histo from a gaussian",100,-3,3);
h1.FillRandom("gaus",10000);
```

`TH1::GetRandom` can be used to return a random number distributed according to the contents of a histogram.

To fill a histogram following the distribution in an existing histogram you can use the second signature of `TH1::FillRandom`.

This code snippet assumes that `h` is an existing histogram (`TH1`).

```cpp
root [] TH1F h2("h2","Random Histo",100,-3,3);
root [] h2->FillRandom(h,1000);
```

The distribution contained in the histogram `h` (`TH1`) is integrated over the channel contents. It is normalized to 1. Getting one random number implies:

- Generating a random number between 0 and 1 (say \( r_1 \))
- Find the bin in the normalized integral for \( r_1 \)
- Fill histogram channel

The second parameter (1000) indicates how many random numbers are generated.

Adding, Dividing, and Multiplying

Many types of operations are supported on histograms or between histograms:

- Addition of a histogram to the current histogram
- Additions of two histograms with coefficients and storage into the current histogram
- Multiplications and Divisions are supported in the same way as additions.
- The Add, Divide and Multiply functions also exist to add, divide or multiply a histogram by a function.

If a histogram has associated error bars (`TH1::Sumw2` has been called), the resulting error bars are also computed assuming independent histograms. In case of divisions, binomial errors are also supported.

Projections

One can:

- Make a 1-D projection of a 2-D histogram or Profile. See functions `TH2::ProjectionX`, `TH2::ProjectionY`, `TH2::ProfileX`, `TH2::ProfileY`, `TPProfile::ProjectionX`, `TPProfile::ProjectionY`.
- Make a 1-D, 2-D or profile out of a 3-D histogram see functions `TH3::ProjectionX`, `TH3::ProjectionY`, `TH3::ProjectionZ`.
- Make these projections via: `TH2::FitSlicesX`, `TH2::FitSlicesY`, `TH3::FitSlicesZ`.

Drawing Histograms

When you call the `Draw` method of a histogram (`TH1::Draw`) for the first time, it creates a `THistPainter` object and saves a pointer to painter as a data member of the histogram. The `THistPainter` class specializes in the drawing of histograms. It is separate from the histogram so that one can have histograms without the graphics overhead, for example in a batch program. The choice to give each histogram its own painter rather than a central singleton painter, allows two histograms to be drawn in two threads without overwriting the painter's values.

When a displayed histogram is filled again, you do not have to call the Draw method again. The image is refreshed the next time the pad is updated. A pad is updated after one of these three actions:

- A carriage control on the ROOT command line
- A click inside the pad
- A call to `TPad::Update`

By default, a call to `TH1::Draw` clears the pad of all objects before drawing the new image of the histogram. You can use the "SAME" option to leave the previous display in tact and superimpose the new histogram. The same histogram can be drawn with different graphics options in different pads.

When a displayed histogram is deleted, its image is automatically removed from the pad.

To create a copy of the histogram when drawing it, you can use `TH1::DrawClone`. This will clone the histogram and allow you to change and delete the original one without affecting the clone.

Setting the Style

Histograms use the current style `gStyle`, which is the global object of class `TStyle`. To change the current style for histograms, the `TStyle` class provides a multitude of methods ranging from setting the fill color to the axis tick marks. Here are a few examples:
void SetHistFillColor(Color_t color = 1)
void SetHistFillStyle(Style_t styl = 0)
void SetHistLineColor(Color_t color = 1)
void SetHistLineStyle(Style_t styl = 0)
void SetHistLineWidth(Width_t width = 1)

When you change the current style and would like to propagate the change to a previously created histogram you can call TH1::UseCurrentStyle. You will need to call UseCurrentStyle on each histogram.

When reading many histograms from a file and you wish to update them to the current style you can use gROOT::ForceStyle and all histograms read after this call will be updated to use the current style (also see the chapter Graphics and Graphic User Interfaces).

When a histogram is automatically created as a result of a TTree::Draw, the style of the histogram is inherited from the tree attributes and the current style is ignored. The tree attributes are the ones set in the current TStyle at the time the tree was created. You can change the existing tree to use the current style, by calling TTree::UseCurrentStyle().

The following draw options are supported on all histogram classes:

- "AXIS": Draw only the axis
- "HIST": Draw only the histogram outline (if the histogram has errors, they are not drawn)
- "SAME": Superimpose on previous picture in the same pad
- "CYL": Use cylindrical coordinates
- "POL": Use polar coordinates
- "SPH": Use spherical coordinates
- "PSR": Use pseudo-rapidity/phi coordinates
- "LEGO": Draw a lego plot with hidden line removal
- "LEGO1": Draw a lego plot with hidden surface removal
- "LEGO2": Draw a lego plot using colors to show the cell contents
- "SURF": Draw a surface plot with hidden line removal
- "SURF1": Draw a surface plot with hidden surface removal
- "SURF2": Draw a surface plot using colors to show the cell contents
- "SURF3": Same as SURF with a contour view on the top
- "SURF4": Draw a surface plot using Gouraud shading

The following options are supported for 1-D histogram classes:

- "AH": Draw the histogram, but not the axis labels and tick marks
- "B": Draw a bar chart
- "C": Draw a smooth curve through the histogram bins
- "E": Draw the error bars
- "E0": Draw the error bars including bins with 0 contents
- "E1": Draw the error bars with perpendicular lines at the edges
- "E2": Draw the error bars with rectangles
- "E3": Draw a fill area through the end points of the vertical error bars
- "E4": Draw a smoothed filled area through the end points of the error bars
- "L": Draw a line through the bin contents
- "P": Draw a (Poly) marker at each bin using the histogram's current marker style
- "P0": Draw current marker at each bin including empty bins
- "PH": Draw histogram with a " at each bin
- "LF": Draw histogram as with option "L" but with a fill area. Note that "L" also draws a fill area if the hist fillcolor is set but the fill area corresponds to the histogram contour.
- "B": Force histogram to be drawn in high resolution mode. By default, the histogram is drawn in low resolution in case the number of bins is greater than the number of pixels in the current pad.

The following options are supported for 2-D histogram classes:

- "ARR": Arrow mode. Shows gradient between adjacent cells
- "BOX": Draw a box for each cell with surface proportional to contents
- "COL": Draw a box for each cell with a color scale varying with contents
Histograms

October 2002 - version 3.02c

---

Statistics Display

By default, drawing a histogram includes drawing the statistics box. To eliminate the statistics box use: \texttt{TH1::SetStats(kFALSE)}.

If the statistics box is drawn, you can select the type of information displayed with \texttt{gStyle\rightarrow SetOptStat(mode)}. The mode has up to seven digits that can be set to one of (1) or off (0). Mode = \texttt{1} (default = 0001111)

- \(n = 1\) the name of histogram is printed
- \(e = 1\) the number of entries printed
- \(m = 1\) the mean value printed
- \(r = 1\) the root mean square printed

---

Setting Tick Marks on the Axis

The \texttt{TPad::SetTicks(tx,ty)} method specifies the type of tick marks on the axis. Assume \(tx = \texttt{gPad\rightarrow GetTickx()}\) and \(ty = \texttt{gPad\rightarrow GetTicky()}\).

- \(tx = 1\): tick marks on top side are drawn (inside)
- \(tx = 2\): tick marks and labels on top side are drawn
- \(ty = 1\): tick marks on right side are drawn (inside)
- \(ty = 2\): tick marks and labels on right side are drawn

By default only the left Y axis and X bottom axis are drawn (\(tx\) \= \(ty\) \= \(0\) ). To see tick marks on top side are drawn (inside) use \(tx = 1\); tick marks on top side are drawn (inside) use \(tx = 2\).

Use \texttt{TPad::SetTicks(tx,ty)} to set these options. See also \texttt{The TAxis methods to set specific axis attributes}. In case multiple color filled histograms are drawn on the same pad, the fill area may hide the axis tick marks. One can force a redraw of the axis over all the histograms by calling:

\texttt{gPad\rightarrow RedrawAxis();}

---

Giving Titles to the X, Y and Z Axis

Because the axis title is an attribute of the axis, you have to get the axis first and then call \texttt{TAxis::SetTitle}. By default only the left Y axis and X bottom axis are drawn. The title can be any \texttt{TLatex} string. The titles are part of the persistent histogram. For example if you wanted to write \(E\) with a subscript (\(T\)) you could use this:

\texttt{h\rightarrow GetXaxis()\rightarrow SetTitle("X axis title");}
\texttt{h\rightarrow GetYaxis()\rightarrow SetTitle("Y axis title");}

The histogram title and the axis titles can be any \texttt{TLatex} string. The titles are part of the persistent histogram. For example if you wanted to write \(E\) with a subscript (\(T\)) you could use this:

\texttt{h\rightarrow GetXaxis()\rightarrow SetTitle("X axis title");}
\texttt{h\rightarrow GetYaxis()\rightarrow SetTitle("Y axis title");}

---

Setting Line, Fill, Marker, and Text Attributes

The histogram classes inherit from the attribute classes: \texttt{TAttLine}, \texttt{TAttFill}, \texttt{TAttMarker} and \texttt{TAttText}. See the description of these classes for the list of options.
h->GetXaxis()->SetTitle("E_{T}");

For a complete explanation of the LaTeX mathematical expressions see chapter "Graphics and Graphical User Interface".

The SCATter Plot Option

By default, 2D histograms are drawn as scatter plots. For each cell \((i,j)\) a number of points proportional to the cell content are drawn. A maximum of 500 points per cell are drawn. If the maximum is above 500 contents are normalized to 500.

The ARRow Option

The ARR option shows the gradient between adjacent cells. For each cell \((i,j)\) an arrow is drawn. The orientation of the arrow follows the cell gradient.

The BOX Option

For each cell \((i,j)\) a box is drawn with surface proportional to contents.

The ERRor Bars Options

- 'E' Default. Draw only the error bars, without markers
- 'E0' Draw also bins with 0 contents
- 'E1' Draw small lines at the end of the error bars
- 'E2' Draw error rectangles
- 'E3' Draw a fill area through the end points of the vertical error bars
- 'E4' Draw a smoothed filled area through the end points of the error bars.

The COLor Option

For each cell \((i,j)\) a box is drawn with a color proportional to the cell content. The color table used is defined in the current style (gStyle). The color palette in TStyle can be modified with TStyle::SetPalette.
The TEXT Option

For each cell (i, j) the cell content is printed. The text attributes are:

- Text font = current TStyle font
- Text size = 0.02 * pad-height * marker-size
- Text color = marker color

The CONTour Options

The following contour options are supported:

- "CONT": Draw a contour plot (same as CONT0)
- "CONT0": Draw a contour plot using surface colors to distinguish contours
- "CONT1": Draw a contour plot using line styles to distinguish contours
- "CONT2": Draw a contour plot using the same line style for all contours
- "CONT3": Draw a contour plot using fill area colors
- "CONT4": Draw a contour plot using surface colors (SURF option at theta = 0)

The default number of contour levels is 20 equidistant levels and can be changed with TH1::SetContour.

When option "LIST" is specified together with option "CONT", the points used to draw the contours are saved in the TGraph object and are accessible in the following way:

TObjArray* contours = gROOT->GetListOfSpecials()->FindObject("contours");
Int_t ncontours = contours->GetSize();
TList* list = (TList*)contours->At(i);

Where "i" is a contour number and list contains a list of TGraph objects. For one given contour, more than one disjoint poly-line may be generated. The number of TGraphs per contour is given by list->GetSize().

Here we show how to access the first graph in the list:

TGraph* gr1 = (TGraph*)list->First();
The LEGO Options

In a lego plot, the cell contents are drawn as 3-d boxes, with the height of the box proportional to the cell content. A lego plot can be represented in several coordinate systems; the default system is Cartesian coordinates. Other possible coordinate systems are CYL, POL, SPH, and PSR.

- "LEGO": Draw a lego plot with hidden line removal
- "LEGO1": Draw a lego plot with hidden surface removal
- "LEGO2": Draw a lego plot using colors to show the cell contents

See TStyle::SetPalette to change the color palette. We suggest you use palette 1 with the call:

```gStyle->SetColorPalette(1);```

The SURFace Options

In a surface plot, cell contents are represented as a mesh. The height of the mesh is proportional to the cell content. A surface plot can be represented in several coordinate systems. The default is Cartesian coordinates, and the other possible systems are CYL, POL, SPH, and PSR.

- "SURF": Draw a surface plot with hidden line removal
- "SURF1": Draw a surface plot with hidden surface removal
- "SURF2": Draw a surface plot using colors to show the cell contents
- "SURF3": Same as SURF with a contour view on the top
- "SURF4": Draw a surface plot using Gouraud shading

The following picture uses SURF1. See TStyle::SetPalette to change the color palette. We suggest you use palette 1 with the call:

```gStyle->SetColorPalette(1);```
The BAR options

When the option "bar" or "hbar" is specified, a bar chart is drawn.

**Vertical BAR chart**

The options are "bar", "bar0", "bar1", "bar2", "bar3", "bar4".

- The bar is filled with the histogram fill color.
- The left side of the bar is drawn with a light fill color.
- The right side of the bar is drawn with a dark fill color.
- The percentage of the bar drawn with either the light or dark color is:
  - 00 per cent for option "bar" or "bar0"
  - 10 per cent for option "bar1"
  - 20 per cent for option "bar2"
  - 30 per cent for option "bar3"
  - 40 per cent for option "bar4"

Use `TH1::SetBarWidth` to control the bar width (default is the bin width)
Use `TH1::SetBarOffset` to control the bar offset (default is 0)
See example in `$ROOTSYS/tutorials/hbars.C`

**Horizontal BAR chart:**

The options for the horizontal bar chart are:
"hbar", "hbar0", "hbar1", "hbar2", "hbar3", "hbar4"

- A horizontal bar is drawn for each bin.
- The bar is filled with the histogram fill color.
- The bottom side of the bar is drawn with a light fill color.
- The top side of the bar is drawn with a dark fill color.
- The percentage of the bar drawn with either the light or dark color is:
  - 0 per cent for option "hbar" or "hbar0"
  - 10 per cent for option "hbar1"
  - 20 per cent for option "hbar2"
  - 30 per cent for option "hbar3"
  - 40 per cent for option "hbar4"

Use `TH1::SetBarWidth` to control the bar width (default is the bin width)
Use `TH1::SetBarOffset` to control the bar offset (default is 0)
See example in `$ROOTSYS/tutorials/hbars.C`

**The Z Option: Display the Color Palette on the Pad**

The "Z" option can be specified with the options: BOX, COL, CONT, SURF, and LEGO to display the color palette with an axis indicating the value of the corresponding color on the right side of the picture.

If there is not enough space on the right side, you can increase the size of the right margin by calling `TPad::SetRightMargin`.
The attributes used to display the palette axis values are taken from the Z axis of the object. For example, you can set the labels size on the palette axis with:

\[ \text{hist->GetZaxis()->SetLabelSize().} \]

### Setting the color palette

You can set the color palette with `TStyle::SetPalette`, e.g.

\[ \text{gStyle->SetPalette(ncolors,colors);} \]

For example, the option `COL` draws a 2-D histogram with cells represented by a box filled with a color index, which is a function of the cell content. If the cell content is N, the color index used will be the color number in `colors[N]`. If the maximum cell content is greater than `ncolors`, all cell contents are scaled to `ncolors`.

If `ncolors <= 0`, a default palette (see below) of 50 colors is defined. This palette is recommended for pads, labels.

If `ncolors == 1` && `colors == 0`, a pretty palette with a violet to red spectrum is created. We recommend you use this palette when drawing lego plots, surfaces, or contours.

If `ncolors > 0` and `colors == 0`, the default palette is used with a maximum of `ncolors`.

The default palette defines:

- Index 0 to 9: shades of gray
- Index 10 to 19: shades of brown
- Index 20 to 29: shades of blue
- Index 30 to 39: shades of red
- Index 40 to 49: basic colors

The color numbers specified in the palette can be viewed by selecting the item "colors" in the "VIEW" menu of the canvas toolbar. The color's red, green, and blue values can be changed via `TColor::SetRGB`.

---

### Drawing a Sub-range of a 2-D Histogram (the [cutg] Option)

Using a `TCutG` object, it is possible to draw a sub-range of a 2-D histogram. One must create a graphical cut (mouse or C++) and specify the name of the cut between `[]` in the Draw option.

For example, with a `TCutG` named "cutg", one can call:

\[ \text{myHist->Draw("surf1 [cutg]!");} \]

Or, assuming two graphical cuts with name "cut1" and "cut2", one can do:

\[ \text{h1.Draw("lego");} \]
\[ \text{h2.Draw(\"[cut1,-cut2],surf,same\");} \]

The second `Draw` will superimpose on top of the first lego plot a subset of `h2` using the "surf" option with:

- all the bins inside cut1
- all the bins outside cut2

Up to 16 cuts may be specified in the cut string delimited by `"[]"`. Currently only the following drawing options are sensitive to the cuts option: col, box, scat, hist, lego, surf and cartesian coordinates only.

See a complete example in the tutorial `$ROOTSYS/tutorials/fit2a.C`. This tutorial produces the following picture:

---

### Drawing Options for 3-D Histograms

By default a 3-d scatter plot is drawn. If the "BOX" option is specified, a 3-D box with a volume proportional to the cell content is drawn.
Superimposing Histograms with Different Scales

The following script creates two histograms; the second histogram is the bins integral of the first one. It shows a procedure to draw the two histograms in the same pad and it draws the scale of the second histogram using a new vertical axis on the right side.

```c
void twoscales() {
  TCanvas *c1 = new TCanvas("c1","hists with different scales",600,400);
  //create, fill and draw h1
  gStyle->SetOptStat(kFALSE);
  TH1F *h1 = new TH1F("h1","my histogram",100,-3,3);
  Int_t i;
  for (i=0;i<10000;i++) h1->Fill(gRandom->Gaus(0,1));
  h1->Draw();
  c1->Update();
  //create hint1 filled with the bins integral of h1
  TH1F *hint1 = new TH1F("hint1","h1 bins integral",100,-3,3);
  Float_t sum = 0;
  for (i=1;i<=100;i++) {
    sum += h1->GetBinContent(i);
    hint1->SetBinContent(i,sum);
  }
  //scale hint1 to the pad coordinates
  Float_t rightmax = 1.1*hint1->GetMaximum();
  Float_t scale = gPad->GetUymax()/rightmax;
  hint1->SetLineColor(kRed);
  hint1->Scale(scale);
  hint1->Draw("same");
  //draw an axis on the right side
  TAxis *axis = new TAxis(gPad->GetUxmin(),gPad->GetUymin(),
                          gPad->GetUxmax(),gPad->GetUymax(),
                          0,rightmax,510,"+L");
  axis->SetLineColor(kRed);
  axis->SetTextColor(kRed);
  axis->Draw();
}
```

Making a Copy of an Histogram

Like for any other ROOT object derived from TObject, one can use the Clone method. This makes an identical copy of the original histogram including all associated errors and functions:

```c
TH1F *hnew = (TH1F*)h->Clone();
htnew->SetName("hnew");
// renaming is recommended, because otherwise you will
// have 2 histograms with the same name.
```

Normalizing Histograms

You can scale a histogram (TH1 *h) such that the bins integral is equal to the normalization parameter norm with:

```c
Double_t scale = norm/h->Integral();
h->Scale(scale);
```

Saving/Reading Histograms to/from a file

The following statements create a ROOT file and store a histogram on the file. Because TH1 derives from TNamed, the key identifier on the file is the histogram name:

```c
TFile f("histos.root","new");
TH1 h1("hgaus","histo from a gaussian",100,-3,3);
h1.FillRandom("gaus",20000);
h1->Write();
```

To read this histogram in another ROOT session, do:

```c
TFile f("histos.root");
TH1 *h = (TH1*)f.Get("hgaus");
```

One can save all histograms in memory to the file by:

```c
file->Write();
```

For a more detailed explanation, see chapter Input/Output.

Miscellaneous Operations

- TH1::KolmogorovTest(): statistical test of compatibility in shape between two histograms.
- TH1::Smooth(): smoothes the bin contents of a 1-d histogram
- TH1::Integral(): returns the integral of bin contents in a given bin range
- TH1::GetMean(int axis): returns the mean value along axis
- TH1::GetRMS(int axis): returns the Root Mean Square along axis
- TH1::GetEntries (): returns the number of entries
- TH1::Reset(): resets the bin contents and errors of a histogram

Alphanumeric Bin Labels

By default, a histogram axis is drawn with its numeric bin labels. One can specify alphanumeric labels instead.

**Option 1: SetBinLabel**

To set an alphanumeric bin label call:

```c
TAxis::SetBinLabel(bin, label);
```

This can always be done before or after filling. When the histogram is drawn, bin labels will be automatically drawn.

**Option 2: Fill**

You can also call a `fill` function with one of the arguments being a string:

```c
hist1->Fill(somename, weight);
hist2->Fill(x, somename, weight);
hist2->Fill(somename, y, weight);
hist2->Fill(somenamex, somenameny, weight);
```

See example in `$ROOTSYS/tutorials/hlabels1.C, hlabels2.C`.

**Option 3: TTree::Draw**

You can use a `char*` variable type to histogram strings with TTree::Draw.

```c
tree.Draw("Nation::Division");
```

Where "Nation" and "Division" are two `char*` branches of a `Tree`.

There is an example in `$ROOTSYS/tutorials/cernstaff.C`.

If a variable is defined as `char*` it is drawn as a string by default. You change that and draw the value of `char[0]` as an integer by adding an arithmetic operation to the expression as shown below.

```c
tree.Draw("MyChar + 0");
```

Where `MyChar` is `char[5]`

Sort Options

When using the options 2 or 3 above, the labels are automatically added to the list (THashList) of labels for a given axis. By default, an axis is drawn with the order of bins corresponding to the filling sequence. It is possible to reorder the axis alphabetically or by increasing or decreasing values.

The reordering can be triggered via the TAxis context menu by selecting the menu item "LabelsOption" or by calling directly.

```c
TH1::LabelsOption(option, axis)
```

Where axis may be "X", "Y" or "Z"

**option** may be:

- "a" sort by alphabetic order
- "x" sort by decreasing values
- "y" sort by increasing values
- "h" draw labels horizontal
- "v" draw labels vertical
- "u" draw labels up (end of label right adjusted)
- "d" draw labels down (start of label left adjusted)
When using the option 2 above, new labels are added by doubling the current number of bins in case one label does not exist yet. When the Filling is terminated, it is possible to trim the number of bins to match the number of active labels by calling:

```c
TH1::LabelsDeflate(axis)
```

Where axis = "X", "Y" or "Z"

This operation is automatic when using TTree::Draw.

Once bin labels have been created, they become persistent if the histogram is written to a file or when generating the C++ code via SavePrimitive.

**Histogram Stacks**

A THStack is a collection of TH1 (or derived) objects. To add a histogram to the stack use THStack::Add(TH1 *h). The THStack owns the objects in the list.

![Histogram Stack Example](image)

By default, THStack::Draw draws the histograms stacked as shown in the left pad in the picture above.

If the option "nostack" is used, the histograms are superimposed as if they were drawn one at a time using the "same" draw option. The right pad in the picture above illustrates the THStack drawn with the "nostack" option.

```c
hs->Draw("nostack");
```

**Profile Histograms**

Profile histograms are in many cases an elegant replacement of two-dimensional histograms. The relationship of two quantities X and Y can be visualized by a two-dimensional histogram or a scatter-plot. Its representation is not particularly satisfactory, except for sparse data. If Y is an unknown [but single-valued] function of X, it can be displayed by a profile histogram with much better precision than by a scatter-plot. Profile histograms display the mean value of Y and its RMS for each bin in X.

The following shows the contents [capital letters] and the values shown in the graphics [small letters] of the elements for bin j.

When you fill a profile histogram with TProfile.Fill[x,y];
E[j] will contain for each bin j the sum of the y values for this bin
L[j] contains the number of entries in the bin J
h[j] = H[j] / L[j]
s[j] = sqrt(E[j] / L[j] - h[j]**2)
e[j] = s[j] / sqrt(L[j])

In the special case where s[j] is zero, when there is only one entry per bin, e[j] is computed from the average of the s[j] for all bins. This approximation is used to keep the bin during a fit operation.
The TProfile Constructor

The TProfile constructor takes up to six arguments. The first five parameters are similar to TH1D::TH1D.

```
TProfile(const char *name,const char *title,Int_t nbins,Axis_t xlow,Axis_t xup,Option_t *option)
```

The first five parameters are similar to TH1D::TH1D. All values of \( y \) are accepted at filling time. To fill a profile histogram, you must use TProfile::Fill function.

Note that when filling the profile histogram the method TProfile::Fill checks if the variable \( y \) is between \( fYmin \) and \( fYmax \). If a minimum or maximum value is set for the \( Y \) scale before filling, then all values below \( ymin \) or above \( ymax \) will be discarded. Setting the minimum or maximum value for the \( Y \) scale before filling has the same effect as calling the special TProfile constructor above where \( ymin \) and \( ymax \) are specified.

Build Options

The last parameter is the build option. If a bin has \( N \) data points all with the same value \( Y \), which is the case when dealing with integers, the spread in \( Y \) for that bin is zero, and the uncertainty assigned is also zero, and the bin is ignored in making subsequent fits. If \( SQRT(Y) \) was the correct error in the case above, then \( SQRT(Y)/SQRT(N) \) would be the correct error here. In fact, any bin with non-zero number of entries \( N \) but with zero spread should have an uncertainty \( SQRT(Y)/SQRT(N) \).

Now, is \( SQRT(Y)/SQRT(N) \) really the correct uncertainty? That it is only in the case where the \( Y \) variable is some sort of counting statistics, following a Poisson distribution. This is the default case. However, \( Y \) can be any variable from an original NTUPLE, and does not necessarily follow a Poisson distribution.

The computation of errors is based on the parameter option:

- `'` Errors are:
  - \( spread/SQRT(N) \) for a non-zero spread
  - \( SQRT(Y)/SQRT(N) \) for a spread of zero and some data points
  - 0 for no data points

- `'` Errors are:
  - \( spread \) for a non-zero spread
  - \( SQRT(Y) \) for a spread of zero and some data points
  - 0 for no data points

- `'` Errors are:
  - \( spread/SQRT(N) \) for a non-zero spread
  - \( 1/SQRT(12*N) \) for a Spread of zero and some data points
  - 0 for no data points

- `'` Errors are:
  - \( spread/SQRT(N) \) for a non-zero spread
  - \( sigma/SQRT(N) \) for a spread of zero and some data points
  - 0 for no data points

The third case (option `'`) is used for integer \( Y \) values with the uncertainty of \( +0.5 \), assuming the probability that \( Y \) takes any value between \( Y-0.5 \) and \( Y+0.5 \) is uniform (the same argument for \( Y \) uniformly distributed between \( Y \) and \( Y+1 \)). An example is an ADC measurement.

Example of a TProfile

Here is a simple example of a profile histogram with its graphic output:

```cpp
// Create a canvas giving the coordinates and the size
TCanvas *c1 = new TCanvas
("c1","Profile example",200,10,700,500);

// Create a profile with the name, title, the number of
// bins, the low and high limit of the x-axis and the low
// and high limit of the y-axis. No option is given so
// the default is used.
hprof = new TProfile
("hprof","Profile of pz versus px",100,-4,4,0,20);

// Fill the profile 25000 times with random numbers
Float_t px, py, pz;
for ( Int_t i=0; i<25000; i++) {
  // Use the random number generator to get two
  // numbers following a gaussian distribution
  // with mean=0 and sigma=1
  gRandom->Rannor(px,py);
  pz = px*px + py*py;
  hprof->Fill(px,pz,1);
}

hprof->Draw();
```

The `'G` option is useful, if all \( Y \) variables are distributed according to some known Gaussian of standard deviation \( Sigma \). For example when all \( Y \)'s are experimental quantities measured with the same instrument with precision \( Sigma \).
Drawing a Profile without Error Bars

To draw a profile histogram and not show the error bars use the "HIST" option in the TProfile::Draw method. This will draw the outline of the TProfile.

Create a Profile from a 2D Histogram

You can make a profile from a histogram using the methods TH2::ProfileX and TH2::ProfileY.

Create a Histogram from a Profile

To create a regular histogram from a profile histogram, use the method TProfile::ProjectionX. This example instantiates a TH1D object by copying the TH1D piece of a TProfile.

```
TH1D *sum = myProfile.ProjectionX();
```

You can do the same with a 2D profile with the TProfile2D::ProjectionXY method.

Generating a Profile from a TTree

The 'prof' and 'profs' options in the TTree::Draw method (see the chapter on Trees) generate a profile histogram (TProfile), given a two dimensional expression in the tree, or a TProfile2D given a three dimensional expression.

Note that you can specify 'prof' or 'profs': 'prof' generates a TProfile with error on the mean, 'profs' generates a TProfile with error on the spread.

2D Profiles

The class for a 2D Profile is called TProfile2D. It is in many cases an elegant replacement of a three-dimensional histogram. The relationship of three measured quantities X, Y and Z can be visualized by a three-dimensional histogram or scatter-plot; its representation is not particularly satisfactory, except for sparse data. If Z is an unknown (but single-valued) function of X, Y, it can be displayed with a TProfile2D with better precision than by a scatter-plot.

A TProfile2D displays the mean value of Z and its RMS for each cell in X, Y. The following shows the cumulated contents (capital letters) and the values displayed (small letters) of the elements for cell I, J.

When you fill a profile histogram with TProfile2D::Fill(x,y,z):

- E[i,j] will contain for each bin i,j the sum of the z values for this bin
- L[i,j] contains the number of entries in the bin i,j
- s[i,j] or e[i,j] will be the resulting error depending on the selected option described in Build Options above.

- E[i,j] = sum z
- L[i,j] = sum i
- H[i,j] = E[i,j]/L[i,j]
- s[i,j] = sqrt(E[i,j]/L[i,j]) - H[i,j]**2
- e[i,j] = s[i,j]/sqrt(L[i,j])

In the special case where s[i,j] is zero, when there is only one entry per cell, e[i,j] is computed from the average of the s[i,j] for all cells. This approximation is used to keep the cell during a fit operation.

Example of a TProfile2D histogram

```c
// Creating a Canvas and a TProfile2D
TCanvas *c1 = new TCanvas
("c1","Profile histogram example",200,10,700,500); 
hprof2d = new TProfile2D
("hprof2d","Profile of pz versus px and py", 
 40,-4,4,40,-4,4,0,20);

// Filling the TProfile2D with 25000 points
Float_t px, py, pz;
for (Int_t i=0; i<25000; i++) {
  gRandom->Rannor(px,py);
  pz = px*px + py*py;
  hprof2d->Fill(px,py,pz,1);
}
hprof2d->Draw();
```
4 Graphs

A graph is a graphics object made of two arrays X and Y, holding the x, y coordinates of n points. There are several graph classes, they are: TGraph, TGraphErrors, TGraphAsymmErrors, and TMultigraph.

TGraph

The TGraph class supports the general case with non equidistant points, and the special case with equidistant points.

Creating Graphs

Graphs are created with the constructor. Here is an example. First we define the arrays of coordinates and then create the graph. The coordinates can be arrays of doubles or floats.

```c
Int_t n = 20;
Double_t x[n], y[n];
for (Int_t i=0;i<n;i++) {
x[i] = i*0.1;
y[i] = 10*sin(x[i]+0.2);
}
TGraph * gr1 = new TGraph (n, x, y);
```

An alternative constructor takes only the number of points (n). It is expected that the coordinates will be set later.

```c
TGraph * gr2 = new TGraph(n);
```

Graph Draw Options

The various draw options for a graph are explained in TGraph::PaintGraph. They are:

- "L" A simple poly-line between every points is drawn
- "F" A fill area is drawn
- "A" Axis are drawn around the graph
- "C" A smooth curve is drawn
- "*" A star is plotted at each point
- "P" The current marker of the graph is plotted at each point
- "B" A bar chart is drawn at each point
- "[]" Only the end vertical/horizontal lines of the error bars are drawn. This option only applies to the TGraphAsymmErrors.

The options are not case sensitive and they can be concatenated in most cases. Let’s look at some examples.

Continuous line, Axis and Stars (AC*)

```c
{ 
    Int_t n = 20;
    Double_t x[n], y[n];
    for (Int_t i=0;i<n;i++) {
        x[i] = i*0.1;
        y[i] = 10*sin(x[i]+0.2);
    }
    // create graph
    TGraph *gr = new TGraph(n,x,y);
    TCanvas *c1 = new TCanvas ("c1","Graph Draw Options", 200, 10, 600, 400);
    // draw the graph with axis,continuous line, and
    // put a * at each point
    gr->Draw("AC*");
}
```
Bar Graphs (AB)

```
root [] TGraph *gr1 = new TGraph(n,x,y);
gr1->SetFillColor(40);
gr1->Draw("AB");
```

This code will only work if n, x, and y is defined. The previous example defines these.
You need to set the fill color, because by default the fill color is white and will not be visible on a white canvas. You also need to give it an axis, or the bar chart will not be displayed properly.

Filled Graphs (AF)

```
root [] TGraph *gr3 = new TGraph(n,x,y);
gr3->SetFillColor(45);
gr3->Draw("AF");
```

This code will only work if n, x, and y is defined. The first example defines these.
You need to set the fill color, because by default the fill color is white and will not be visible on a white canvas. You also need to give it an axis, or the bar chart will not be displayed properly.

Currently one can not specify the "CF" option.
Superimposing two Graphs

To super impose two graphs you need to draw the axis only once, and leave out the "A" in the draw options for the second graph. Here is an example:

```c
int n = 20;
double x[n], y[n], x1[n], y1[n];
// create the blue graph with a cos function
for (int i = 0; i < n; i++) {
    x[i] = i*0.5;
    y[i] = 5*cos(x[i]+0.2);
    x1[i] = i*0.5;
    y1[i] = 5*sin(x[i]+0.2);
}
tgraph *gr1 = new tgraph(n, x, y);
tgraph *gr2 = new tgraph(n, x1, y1);
tcanvas *c1 = new tcanvas("c1","Two Graphs" , 200,
                         10, 600, 400);
// draw the graph with axis, continuous line, and
// put a * at each point
gr1->setlinecolor(4);
gr1->draw("AC*");
// superimpose the second graph by leaving out
// the axis option "A"
gr2->setlinecolor(2);
gr2->setmarkerstyle(21);
gr2->draw("CP");
```

TGraphErrors

A TGraphErrors is a TGraph with error bars. The various format options to draw a TGraphErrors are the same for TGraph. In addition, it can be drawn with the "Z" option to leave off the small lines at the end of the error bars.

The constructor has four arrays as parameters. X and Y as in TGraph and X-errors and Y-errors the size of the errors in the x and y direction.

This example is in $ROOTSYS/tutorials/gerrors.C.
TGraphAsymmErrors

A TGraphAsymmErrors is a TGraph with asymmetric error bars. The various format options to draw a TGraphAsymmErrors are as for TGraph.

The constructor has six arrays as parameters. \( X \) and \( Y \) as TGraph and low \( X \)-errors and high \( X \)-errors, low \( Y \)-errors and high \( Y \)-errors. The low value is the length of the error bar to the left and downward, the high value is the length of the error bar to the right and upward.

```cpp
gROOT->Reset();
c1 = new TCanvas("c1","A Simple Graph with error bars", 200,10,700,500);
c1->SetFillColor(42);
c1->SetGrid();
c1->GetFrame()->SetFillColor(21);
c1->GetFrame()->SetBorderSize(12);

// create the arrays for the points
Int_t n = 10;
Double_t x[n]  = {-.22,.05,.25,.35,.5, .61,.7,.85,.89,.95};
Double_t y[n]  = {1,2.9,5.6,7.4,9,9.6,8.7,6.3,4.5,1};

// create the arrays with high and low errors
Double_t exl[n] = {.05,.1,.07,.07,.04,.05,.06,.07,.08,.05};
Double_t eyl[n] = {.8,.7,.6,.5,.4,.4,.5,.6,.7,.8};
Double_t exh[n] = {.02,.08,.05,.03,.03,.04,.05,.06,.03};
Double_t eyh[n] = {.6,.5,.4,.3,.2,.3,.4,.5,.6,.5};

// create TGraphAsymmErrors with the arrays
gr = new TGraphAsymmErrors(n,x,y,exl,exh,eyl,eyh);
gr->SetTitle("TGraphAsymmErrors Example");
gr->SetMarkerColor(4);
gr->SetMarkerStyle(21);
gr->Draw("ALP");
```

TMultiGraph

A TMultiGraph is a collection of TGraph (or derived) objects. Use TMultiGraph::Add to add a new graph to the list. The TMultiGraph owns the objects in the list. The drawing options are the same as for TGraph.

```cpp
// create the points
Int_t n = 10;
Double_t x[n]  = {-.22,.05,.25,.35,.5, .61,.7,.85,.89,.95};
Double_t y[n]  = {1,2.9,5.6,7.4,9,9.6,8.7,6.3,4.5,1};

// create the width of errors in x and y direction
Double_t dx[n] = {-.12,.15,.35,.45,.6,.71,.8,.95,.99,1.05};
Double_t dy[n] = {1,2.9,5.6,7.4,9,9.6,8.7,6.3,4.5,1};

// create two graphs
TGraph *gr1 = new TGraph(n,x2,y2);
TGraphErrors *gr2 = new TGraphErrors(n,x,y,ex,ey);

// create a multigraph and draw it
TMultiGraph *mg = new TMultiGraph();
mg->Add(gr1);
mg->Add(gr2);
mg->Draw("ALP");
```

Fitting a Graph

The Fit method of the graph works the same as the TH1::Fit (see Fitting Histograms).
Setting the Graph’s Axis Title

To give the axis of a graph a title you need to draw the graph first, only then does it actually have an axis object. Once drawn, you set the title by getting the axis and calling the TAxis::SetTitle method, and if you want to center it you can call the TAxis::CenterTitle method.

Assuming that n, x, and y are defined, this code sets the titles of the x and y axes.

```c
root [] gr5 = new TGraph(n,x,y);
root [] gr5->Draw()
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
root [] gr5->Draw("ALP")
root [] gr5->GetXaxis()->SetTitle("X-Axis")
root [] gr5->GetYaxis()->SetTitle("Y-Axis")
root [] gr5->GetXaxis()->CenterTitle()
root [] gr5->GetYaxis()->CenterTitle()
root [] gr5->Draw("ALP")
```

For more graph examples see: these scripts in the $ROOTSYS/tutorials directory graph.C, gerrors.C, zdemo.C, and gerrors2.C.

Zooming a Graph

To zoom a graph you can create a histogram with the desired axis range first. Draw the empty histogram and then draw the graph using the existing axis from the histogram.

The example below is the same graph as above with a zoom in the x and y direction.

```c
{ gROOT->Reset();
c1 = new TCanvas("c1","A Zoomed Graph",200,10,700,500);
// create a histogram for the axis range
hpx = new TH2F
('hpx","Zoomed Graph Example",10, 0.0,10.0,8.0);
// no statistics
hpx->SetStats(kFALSE);
hpx->Draw();

// create a graph
Int_t n = 10;
Double_t x[n] = {-.22,.05,.25,.35,.5,.61,.7,.85,.89,.95};
Double_t y[n] = {1,2.9,5.6,7.4,9,9.6,8.7,6.3,4.5,1};
gr = new TGraph(n,x,y);
gr->SetMarkerColor(4);
gr->SetMarkerStyle(20);
// and draw it without an axis
gr->Draw("LP");
}
```
5 Fitting Histograms

To fit a histogram you can use the Fit Panel on a visible histogram using the GUI, or you can use the TH1::Fit method. The Fit Panel, which is limited, is best for prototyping. The histogram needs to be drawn in a pad before the Fit Panel is available. The TH1::Fit method is more powerful and used in scripts and programs.

The Fit Panel

To display the Fit Panel right click on a histogram to bring up the context menu, then select the menu option: FitPanel.

The first sets of buttons are the predefined functions of ROOT that can be used to fit the histogram. You have a choice of several polynomials, a gaussian, a landau, and an exponential function. You can also define a function and call it "user". It will be linked to the user button on this panel.

You have the option to specify Quiet or Verbose. This is the amount of feedback printed on the root command line on the result of the fit.

When a fit is executed the image of the function is drawn on the current pad. By default the image of the histogram is replaced with the image of the function. Select Same Picture to see the function drawn and the histogram on the same picture.

Select W: Set all weights to 1 to set all errors to 1.
Select E: Compute best errors to use the Minos technique to compute best errors.

When fitting a histogram, the function is attached to the histogram's list of functions. By default the previously fitted function is deleted and replaced with the most recent one, so the list only contains one function. You can select + : Add to list of functions to add the newly fitted function to the existing list of functions for the histogram. Note that the fitted functions are saved with the histogram when it is written to a ROOT file.

By default, the function is drawn on the pad displaying the histogram. Select N: Do not store/draw function to avoid adding the function to the histogram and to avoid drawing it.

Select 0: Do not draw function to avoid drawing the result of the fit.

The Fit Method

To fit a histogram programmatically, you can use the TH1::Fit method. Here is the signature of TH1::Fit and an explanation of the parameters:

```c
void Fit(const char *fname , Option_t *option , Option_t *goption, Axis_t xxmin, Axis_t xxmax)
```

*fname*: The name of the fitted function (the model) is passed as the first parameter. This name may be one of the root's pre-defined function names or a user-defined function.

- gaus: A gaussian function 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:
  \[ f(x) = p0 + p1*x + p2*x^2 + ... \]

- landau: A landau function with mean and sigma. This function has been adapted from the CERNLIB routine G110 denlan.

*option*: The second parameter is the fitting option. Here is the list of fitting options:

- "W" Set all errors to 1
- "I" Use integral of function in bin instead of value at bin center
- "L" Use loglikelihood method (default is chisquare method)
- "U" Use a user specified fitting algorithm
- "Q" Quiet mode (minimum printing)
- "V" Verbose mode (default is between Q and V)
- "E" Perform better errors estimation using Minos technique
- "M" Improve fit results
- "R" Use the range specified in the function range
- "N" Do not store the graphics function, do not draw
- "0" Do not plot the result of the fit. By default the fitted function is drawn unless the option "N" above is specified.
- "+" Add this new fitted function to the list of fitted functions (by default, the previous function is deleted and only the last one is kept)
- "B" Disable the automatic computation of the initial parameter values for the standard functions like poln, expo, and gaus.

*goption*: The third parameter is the graphics option (goption), it is the same as in the TH1::Draw (see Draw Options above).
xxmin, xxmax: The fourth and fifth parameters specify the range over which to apply the fit. By default, the fitting function object is added to the histogram and is drawn in the current pad.

**Fit with a Predefined Function**

To fit a histogram with a predefined function, simply pass the name of the function in the first parameter of `TH1::Fit`. For example, this line fits histogram object `hist` with a gaussian:

```c
root[] hist.Fit("gaus");
```

For pre-defined functions, there is no need to set initial values for the parameters, it is done automatically.

**Fit with a User-Defined Function**

You can create a `TF1` object and use it in the call the `TH1::Fit`. The parameter in to the `Fit` method is the NAME of the `TF1` object.

There are three ways to create a `TF1`.

1. Using C++ like expression using `x` with a fixed set of operators and functions defined in `TFormula`.
2. Same as #1, with parameters.
3. Using a function that you have defined

**Creating a TF1 with a Formula**

Let's look at the first case. Here we call the `TF1` constructor by giving it the formula: `sin(x)/x`.

```c
root[] TF1 *f1 = new TF1("f1", "sin(x)/x", 0,10)
```

You can also use a `TF1` object in the constructor of another `TF1`.

```c
root[] TF1 *f2 = new TF1("f2", "f1 * 2", 0,10)
```

**Creating a TF1 with Parameters**

The second way to construct a `TF1` is to add parameters to the expression. For example, this `TF1` has 2 parameters:

```c
root[] TF1 *f1 = new TF1("f1", "[0]*x*sin([1]*x)", -3, 3);
```

The parameter index is enclosed in square brackets. To set the initial parameters explicitly you can use the `SetParameter` method.

```c
root[] f1->SetParameter(0, 10);
```

This sets parameter 0 to 10. You can also use `SetParameters` to set multiple parameters at once.

```c
root[] f1->SetParameters(10, 5);
```

This sets parameter 0 to 10 and parameter 1 to 5.

We can now draw the `TF1`:

```c
root[] f1->Draw()
```

**Creating a TF1 with a User Function**

The third way to build a `TF1` is to define a function yourself and then give its name to the constructor. A function for a `TF1` constructor needs to have this exact signature:

```c
Double_t fitf(Double_t *x, Double_t *par)
```

The two parameters are:

- `Double_t *x`: a pointer to the dimension array. Each element contains a dimension. For a 1D histogram only `x[0]` is used, for a 2D histogram `x[0]` and `x[1]` is used, and for a 3D histogram `x[0]`, `x[1]`, and `x[2]` are used. For histograms, only 3 dimensions apply, but this method is also used to fit other objects, for example an ntuple could have 10 dimensions.
- `Double_t *par`: a pointer to the parameters array. This array contains the current values of parameters when it is called by the fitting function.

The following script `$ROOTSYS/tutorials/myfit.C` illustrates how to fit a 1D histogram with a user-defined function. First we declare the function.

```c
// define a function with 3 parameters
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;
}
```
Now we use the function:

```c
// this function used fitf to fit a histogram
void fitexample()
{
    // open a file and get a histogram
    TFile* f = new TFile("hsimple.root");
    TH1F* hpx = (TH1F*)f->Get("hpx");
    // create a TF1 object using the function defined above.
    // The last 3 specifies the number of parameters
    // for the function.
    TF1* func = new TF1("fit", fitf, -3, 3, 3);
    // set the parameters to the mean and RMS of the histogram.
    func->SetParameters(500, hpx->GetMean(), hpx->GetRMS());
    // give the parameters meaningful names
    func->SetParNames("Constant", "Mean_value", "Sigma");
    // call TH1::Fit with the name of the TF1 object
    hpx->Fit("fit");
}
```

Fixing and Setting Bounds for Parameters

Parameters must be initialized before invoking the `Fit` method. The setting of the parameter initial values is automatic for the predefined functions: `poln, exp, gaus`. You can disable the automatic computation by specifying the "B" option when calling the `Fit` method.

When a functions is not predefined, the fit parameters must be initialized to some value as close as possible to the expected values before calling the fit function.

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

```c
func->SetParLimits(0, -1, 1);
```

When the lower and upper limits are equal, the parameter is fixed. This statement fixes parameter 4 at 10.

```c
func->SetParameter(4, 10);
func->SetParLimits(4, 77, 77);
```

However, to fix a parameter to 0, one must call the `FixParameter` function:

```c
func->SetParameter(4, 0);
func->FixParameter(4, 0);
```

Note that you are not forced to fix the limits for all parameters. For example, if you fit a function with 8 parameters, you can:

```c
func->SetParameters(0, 3, 1, 1, 0.1, 0.5, 0, 100);
func->SetParLimits(3, -10, 4);
func->FixParameter(4, 0);
```

With this setup, parameters 0 to 2 can vary freely, parameter 3 has boundaries [-10, 4] with initial value -8, and parameter 4 is fixed to 0.

Fitting Sub Ranges

By default, TH1::Fit will fit the function on the defined histogram range. You can specify the option "R" in the second parameter of TH1::Fit to restrict the fit to the range specified in the TF1 constructor. In this example, the fit will be limited to -3 to 3, the range specified in the TF1 constructor.

```c
root[1] TF1* f1 = new TF1("f1","[0]*x*sin([1]*x)*,-3,3");
root[1] hist->Fit("f1", "R");
```

You can also specify a range in the call to TH1::Fit:

```c
root[1] hist->Fit("f1", "R", -2, 2);
```

For more complete examples, see $ROOTSYS/tutorials/myfit.C and $ROOTSYS/tutorials/multifit.C.

Example: Fitting Multiple Sub Ranges

The script for this example is in $ROOTSYS/tutorials/multifit.C. It shows how to use several gaussian functions with different parameters on separate sub ranges of the same histogram.

To use a gaussian, or any other ROOT built in function, on a sub range you need to define a new TF1. Each is "derived" from the canned function `gaus`.

```c
// Create 4 TF1 objects, one for each subrange
g1 = new TF1("m1", "gaus", 85, 95);
g2 = new TF1("m2", "gaus", 98, 108);
g3 = new TF1("m3", "gaus", 110, 121);
// The total is the sum of the three, each has three
// parameters.
total = new TF1("mstotal", "gaus(0)+gaus(3)+gaus(6)", 85, 125);
```

Here we fill a histogram with bins defined in the array `x` (see $ROOTSYS/tutorials/multifit.C).

```c
// Create a histogram and set it's contents
h = new TH1F("h1", "Example of several fits in subranges", np, 85, 134);
h->SetMaximum(7);
for (int i=0;i<np;i++)
{
    h->SetBinContent(i+1, x[i]);
}
```

// Define the parameter array for the total function
Double_t par[9];

When fitting simple functions, such as a gaussian, the initial values of the parameters are automatically computed by ROOT. In the more complicated case of the sum of 3 gaussian functions, the initial values of parameters must
be set. In this particular case, the initial values are taken from the result of the individual fits.

The use of the "*" sign is explained below.

```
// fit each function and add it to the list of functions
h->Fit(g1,"R");
h->Fit(g2,"R*");
h->Fit(g3,"R*");
// Get the parameters from the fit
g1->GetParameters(&par[0]);
g2->GetParameters(&par[3]);
g3->GetParameters(&par[6]);
// Use the parameters on the sum
total->SetParameters(par);
h->Fit(total,"R+");
```

### Adding Functions to The List

The example `$ROOTSYS/tutorials/multifit.C` also illustrates how to fit several functions on the same histogram. By default a `Fit` command deletes the previously fitted function in the histogram object. You can specify the option "*" in the second parameter to add the newly fitted function to the existing list of functions for the histogram.

```
root[] hist->Fit("f1","*","",-2,2)
```

Note that the fitted function(s) are saved with the histogram when it is written to a ROOT file.

### Combining Functions

You can combine functions to fit a histogram with their sum. Here is an example, the code is in `$ROOTSYS/tutorials/FitDemo.C`. We have a function that is the combination of a background and lorentzian peak. Each function contributes 3 parameters.

\[
y(E) = a_1 + a_2 E + a_3 E^2 + \text{Ap} \left( \frac{G}{2} \frac{1}{(E-m)^2 + (\frac{G}{2})^2} \right)
\]

- **background**: \(a_1, a_2, a_3\)
- **lorentzianPeak**: \(\text{Ap}, G, m\)

The combination function (`fitFunction`) has six parameters:

```
Double_t background(Double_t *x, Double_t *par) {
    return par[0] + par[1]*x[0] + par[2]*x[0]*x[0];
}
```

```
Double_t lorentzianPeak(Double_t *x, Double_t *par) {
    return (0.5*par[0]*par[1]/TMath::Pi()) /
           TMath::Max( 1.e-10,
                       (x[0]-par[2])*(x[0]-par[2]) + .25*par[1]*par[1]);
}
```

```
Double_t fitFunction(Double_t *x, Double_t *par) {
    return background(x,par) + lorentzianPeak(x,&par[3]);
}
```

This script creates a histogram and fits the combination of the two functions. First we define the two functions and the combination function:

```
// Quadratic background function
Double_t background(Double_t *x, Double_t *par) {
    return par[0] + par[1]*x[0] + par[2]*x[0]*x[0];
}
```

```
// Lorenzian Peak function
Double_t lorentzianPeak(Double_t *x, Double_t *par) {
    return (0.5*par[0]*par[1]/TMath::Pi()) /
           TMath::Max( 1.e-10,
                       (x[0]-par[2])*(x[0]-par[2]) + .25*par[1]*par[1]);
}
```

```
// Sum of background and peak function
Double_t fitFunction(Double_t *x, Double_t *par) {
    return background(x,par) + lorentzianPeak(x,&par[3]);
}
```

//… continued on the next page void FittingDemo() {
```
// improve the picture:
backFcn = new TF1("backFcn",background,0,3,3);
backFcn->SetLineColor(3);

signalFcn = new TF1("signalFcn",lorentzianPeak,0,3,3);
signalFcn->SetLineColor(4);

Double_t par[6];
// writes the fit results into the par array
fitFcn->GetParameters(par);
backFcn->SetParameters(par);

backFcn->Draw("same");
signalFcn->SetParameters(&par[3]);
signalFcn->Draw("same");
}

This is the result:

For another example see:

Associated Function

One or more objects (typically a TF1*) can be added to the list of functions (fFunctions) associated to each histogram. A call to TH1::Fit adds the fitted function to this list. Given a histogram h, one can retrieve the associated function with:

TF1 *myfunc = h->GetFunction("myfunc");

Access to the Fit Parameters and Results

If the histogram (or graph) is made persistent, the list of associated functions is also persistent. Retrieve a pointer to the function with the TH1::GetFunction() method. Then you can retrieve the fit parameters from the function (TF1) with calls such as:

root[] TF1 *fit = hist->GetFunction(function_name);
root[] Double_t chi2 = fit->GetChisquare();
// value of the first parameter
root[] Double_t p1 = fit->GetParameter(0);
// error of the first parameter
root[] Double_t e1 = fit->GetParError(0);

Associated Errors

By default, for each bin, the sum of weights is computed at fill time. One can also call TH1::Sumw2 to force the storage and computation of the sum of the square of weights per bin. If Sumw2 has been called, the error per bin is computed as the sqrt(sum of squares of weights), otherwise the error is set equal to the sqrt(bin content). To return the error for a given bin number, do:

Double_t error = h->GetBinError(bin);

Fit Statistics

You can change the statistics box to display the fit parameters with the TStyle::SetOptFit(mode) method. This mode has four digits.

Mode = pcev (default = 0111)

• v = 1 print name/values of parameters
• e = 1 print errors (if e=1, v must be 1)
• c = 1 print Chi-square/number of degrees of freedom
• p = 1 print probability

For example:

gStyle->SetOptFit(111);

This prints the fit probability, parameter names/values, and errors.
6 A Little C++

This chapter introduces you to some useful insights into C++, to allow you to use of the most advanced features in ROOT. It is in no case a full course in C++.

**Classes, Methods and Constructors**

C++ extends C with the notion of class. If you’re used to structures in C, a class is a struct, that is a group of related variables, which is extended with functions and routines specific to this structure (class). What is the interest?

Consider a struct that is defined this way:

```c
struct Line {
  float x1;
  float y1;
  float x2;
  float y2;
}
```

This structure represents a line to be drawn in a graphical window. \((x_1, y_1)\) are the coordinates of the first point, \((x_2, y_2)\) the coordinates of the second point.

In standard C, if you want to effectively draw such a line, you first have to define a structure and initialize the points (you can try this):

```c
Line firstline;
firstline.x1 = 0.2;
firstline.y1 = 0.2;
firstline.x2 = 0.8;
firstline.y2 = 0.9;
```

This defines a line going from the point \((0.2,0.2)\) to the point \((0.8,0.9)\). To draw this line, you will have to write a function, say `LineDraw(Line l)` and call it with your object as argument:

```c
LineDraw(firstline);
```

In C++, we would not do that. We would instead define a class like this:

```c
class TLine {
  Double_t x1;
  Double_t y1;
  Double_t x2;
  Double_t y2;
  TLine(int x1, int y1, int x2, int y2);
  void Draw();
}
```

Here we added two functions, that we will call methods or member functions, to the `TLine` class. The first method is used for initializing the line objects we would build. It is called a constructor.

The second one is the `Draw` method itself. Therefore, to build and draw a line, we have to do:

```c
TLine l(0.2,0.2,0.8,0.9);
l.Draw();
```

The first line builds the object \(l\) by calling its constructor. The second line calls the `TLine::Draw()` method of this object. You don't need to pass any parameters to this method since it applies to the object \(l\), which knows the coordinates of the line. These are internal variables \(x1, y1, x2, y2\) that were initialized by the constructor.

**Inheritance and Data Encapsulation**

**Inheritance**

We've defined a `TLine` class that contains everything necessary to draw a line. If we want to draw an arrow, is it so different from drawing a line? We just have to draw a triangle at one end. It would be very inefficient to define the class `TArrow` from scratch. Fortunately, inheritance allows a class to be defined from an existing class. We would write something like:

```c
class TArrow : public TLine {
  int ArrowHeadSize;
  void Draw();
  void SetArrowSize(int arrowsize);
}
```

The keyword "public" will be explained later. The class `TArrow` now contains everything that the class `TLine` does, and a couple of things more, the size of the arrowhead and a function that can change it. The Draw method of `TArrow` will draw the head and call the draw method of `TLine`. We just have to write the code for drawing the head!

**Method Overriding**

Giving the same name to a method (remember: method = member function of a class) in the child class (`TArrow`) as in the parent (`TLine`) doesn't give any
problem. This is called \textit{overriding} a method. Draw in \texttt{TArrow} overrides\par Draw in \texttt{TLine}. There is no possible ambiguity since, when one calls the \texttt{Draw()} method, this applies to an object which type is known. Suppose we have an object \texttt{l} of type \texttt{TLine} and an object \texttt{a} of type \texttt{TArrow}. When you want to draw the line, you do:

\begin{verbatim}
l.Draw()
\end{verbatim}

\texttt{Draw()} from \texttt{TLine} is called. If you do:

\begin{verbatim}
a.Draw()
\end{verbatim}

\texttt{Draw()} from \texttt{TArrow} is called and the arrow \texttt{a} is drawn.

\section*{Data Encapsulation}

We have seen previously the keyword "\texttt{public}". This keyword means that every name declared public is seen by the outside world. This is opposed to "\texttt{private}" which means only the class where the name was declared private could see this name. For example, suppose we declare in \texttt{TArrow} the variable \texttt{ArrowHeadSize private}.

\begin{verbatim}
private :
  int ArrowHeadSize;
\end{verbatim}

Then, only the methods (\texttt{member functions}) of \texttt{TArrow} will be able to access this variable. Nobody else will see it. Even the classes that we could derive from \texttt{TArrow} will not see it. On the other hand, if we declare the method \texttt{Draw()} as public, everybody will be able to see it and use it. You see that the character public or private doesn’t depend of the type of argument. It can be a data member, a member function, or even a class.

For example, in the case of \texttt{TArrow}, the base class \texttt{TLine} is declared as public:

\begin{verbatim}
class TArrow : public TLine {
  private :
    int ArrowHeadSize;
  public:
    void Draw();
    void SetArrowSize(int arrowsize);
};
\end{verbatim}

This means that all methods of \texttt{TArrow} will be able to access all methods of \texttt{TLine}, but this will be also true for anybody in the outside world. Of course, this is true provided that \texttt{TLine} accepts the outside world to see its methods/data members. If something is declared private in \texttt{TLine}, nobody will see it, not even \texttt{TArrow} members, even if \texttt{TLine} is declared as a public base class.

What if \texttt{TLine} is declared "\texttt{private}" instead of "\texttt{public}?" Well, it will behave as any other name declared private in \texttt{TArrow}; only the data members and methods of \texttt{TArrow} will be able to access \texttt{TLine}, it’s methods and data members, nobody else.

This may seem a little bit confusing and readers should read a good C++ book if they want more details. Especially since, besides public and private, a member can be protected.

Usually, one puts private the methods that the class uses internally, like some utilities classes, and that the programmer doesn’t want to be seen in the outside world.

With "good" C++ practice (which we have tried to use in ROOT), all data members of a class are private. This is called data encapsulation and is one of the strongest advantages of Object Oriented Programming (OOP). Private data members of a class are not visible, except to the class itself. So, in the outside world, if one wants to access those data members, one should use so called "getters" and "setters" methods, which are special methods used only to get or set the data members. The advantage is that if the programmers want to modify the inner workings of their classes, they can do so without changing what the user sees. The user doesn’t even have to know that something has changed (for the better, hopefully).

For example, in our \texttt{TArrow} class, we would have set the data member \texttt{ArrowHeadSize} private. The setter method is \texttt{SetArrowSize()}, we don’t need a getter method:

\begin{verbatim}
class TArrow : public TLine {
  private :
    int ArrowHeadSize;
  public:
    void Draw();
    void SetArrowSize(int arrowsize);
};
\end{verbatim}

To define an arrow object you call the constructor. This will also call the constructor of \texttt{TLine}, which is the parent class of \texttt{TArrow}, automatically. Then we can call any of the line or arrow public methods such as \texttt{SetArrowSize} and \texttt{Draw}.

\begin{verbatim}
root[] TArrow* myarrow = new TArrow(1,5,89,124);
root[] myarrow->SetArrowSize(10);
root[] myarrow->Draw();
\end{verbatim}

\section*{Creating Objects on the Stack and Heap}

To explain how objects are created on the stack and on the heap we will use the \texttt{Quad} class. You can find the definition in \texttt{$ROOTSYS/tutorials/Quad.h} and \texttt{Quad.cxx}.

The \texttt{Quad} class has four methods. The constructor and destructor, \texttt{Evaluate} which evaluates $ax^2 + bx + c$, and \texttt{Solve} which solves the quadratic equation $ax^2 + bx + c = 0$.

\begin{verbatim}
class Quad {
  public:
    Quad(Float_t a, Float_t b, Float_t c);
    ~Quad();
    Float_t Evaluate(Float_t x) const;
    void Solve() const;
  private:
    Float_t fA;
    Float_t fB;
    Float_t fC;
};
\end{verbatim}

The \texttt{Quad} class has four methods. The constructor and destructor, \texttt{Evaluate} which evaluates $ax^2 + bx + c$, and \texttt{Solve} which solves the quadratic equation $ax^2 + bx + c = 0$.
Quad.cxx:

```c++
#include <iostream.h>
#include <math.h>
#include "Quad.h"

Quad::Quad(Float_t a, Float_t b, Float_t c) {
    fA = a;
    fB = b;
    fC = c;
}

Quad::~Quad() {
    cout << "deleting object with coeffts: "
         << fA << "," << fB << "," << fC << endl;
}

Float_t Quad::Evaluate(Float_t x) const {
    return fA*x*x + fB*x + fC;
}

void Quad::Solve() const {
    Float_t temp = fB*fB - 4.*fA*fC;
    if ( temp > 0. ) {
        temp = sqrt( temp );
        cout << "There are two roots: "
             << (-fB - temp ) / (2.*fA)
             << " and "
             << (-fB + temp ) / (2.*fA)
             << endl;
    } else {
        if ( temp == 0. ) {
            cout << "There are two equal roots: "
                 << -fB / (2.*fA)
                 << endl;
        } else {
            cout << "There are no roots" << endl;
        }
    }
}
```

Let's first look how we create an object. When we create an object by

```cpp
root[] Quad my_object(1.,2.,-3.);
```

We are creating an object on the stack. A FORTRAN programmer may be familiar with the idea; it's not unlike a local variable in a function or subroutine. Although there are still a few old timers who don't know it, FORTRAN is under no obligation to save local variables once the function or subroutine returns unless the SAVE statement is used. If not then it is likely that FORTRAN will place them on the stack and they will "pop off" when the RETURN statement is reached.

To give an object more permanence it has to be placed on the heap.

```cpp
root[] Quad stack_quad(1.,2.,-3.);
```

The second line declares a pointer to Quad called my_objptr. From the syntax point of view, this is just like all the other declarations we have seen so far, i.e. this is a stack variable. The value of the pointer is set equal to new Quad(1., 2., -3.);

new, despite its looks, is an operator and creates an object or variable of the type that comes next. Quad in this case, on the heap. Just as with stack objects it has to be initialized by calling its constructor. The syntax requires that the argument list follow the type. This one statement has brought two items into existence, one on the heap and one on the stack. The heap object will live until the delete operator is applied to it.

There is no FORTRAN parallel to a heap object; variables either come and go as control passes in and out of a function or subroutine, or, like a COMMON block variables, live for the lifetime of the program. However, most people in HEP who use FORTRAN will have experience of a memory manager and the act of creating a bank is a good equivalent of a heap object. For those who know systems like ZEBRA, it will come as a relief to learn that objects don't move, C++ does not garbage collect, so there is never a danger that a pointer to an object becomes invalid for that reason. However, having created an object, it is the user's responsibility to ensure that it is deleted when no longer needed, or to pass that responsibility onto to some other object. Failing to do that will result in a memory leak, one of the most common and most hard-to-find C++ bugs.

To send a message to an object via a pointer to it, you need to use the "->" operator e.g.:

```cpp
root[] my_objptr->Solve();
```

Although we chose to call our pointer my_objptr, to emphasize that it is a pointer, heap objects are so common in an OO program that pointer names rarely reflect the fact - you have to be careful that you know if you are dealing with an object or its pointer! Fortunately, the compiler won't tolerate an attempt to do something like:

```cpp
root[] my_objptr.Solve();
```

Although this is a permitted by the CINT shortcuts, it is one that you are strongly advised not to follow!

As we have seen, heap objects have to be accessed via pointers, whereas stack objects can be accessed directly. They can also be accessed via pointers:

```cpp
root[] my_objptr->Solve();
```

Here we have a Quad pointer that has been initialized with the address of a stack object. Be very careful if you take the address of stack objects. As we shall see soon, they get deleted automatically, which could leave you with an illegal pointer. Using it will corrupt and may well crash the program!

It is time to look at the destruction of objects. Just as its constructor is called when it is created, so its destructor is called when it is destroyed. The compiler will provide a destructor that does nothing if none is provided. We will add one to our Quad class so that we can see when it gets called.

The destructor is named by the class but with the prefix ~ which is the C++ one's complement i.e. bit wise complement, and hence has destruction overtones! We declare it in the .h file and define it in the .cxx file. It does not do much except print out that it has been called (still a useful debug technique despite today's powerful debuggers!). Now run root, load the Quad
class and create a heap object:

```cpp
class Quad
{
  public:
    Quad(): a(1.0), b(2.0), c(-3.0) {}

  private:
    double a, b, c;
};
```

To delete the object:

```cpp
delete my_objptr;
my_objptr = 0;
```

You should see the print out from its destructor. Setting the pointer to zero afterwards isn't strictly necessary (and CINT does it automatically), but the object is no more, and any attempt to use the pointer again will, as has already been stated, cause grief.

So much for heap objects, but how do stack objects get deleted? In C++ a stack object is deleted as soon as control leaves the innermost compound statement that encloses it. So it is singularly futile to do something like:

```cpp
{  Quad my_object(1.,2.,-3.); }
```

CINT does not follow this rule; if you type in the above line you will not see the destructor message. As explained in the Script lesson, you can load in compound statements, which would be a bit pointless if everything disappeared as soon as it was loaded! Instead, to reset the stack you have to type:

```cpp
gROOT->Reset();
```

This sends the Reset message via the global pointer to the ROOT object, which, amongst its many roles, acts as a resource manager. Start ROOT again and type in the following:

```cpp
Quad my_object(1.,2.,-3.);
Quad* my_objptr = new Quad(4., 5., -6.);
gROOT->Reset();
```

You will see that this deletes the first object but not the second. We have also painted ourselves into a corner, as `my_objptr` was also on the stack. This command will fail.

```cpp
my_objptr->Solve();
```

CINT no longer knows what `my_objptr` is. This is a great example of a memory leak: the heap object exists but we have lost our way to access it. In general, this is not a problem. If any object will outlive the compound statement in which it was created then it will be pointed to by a more permanent pointer, which frequently is part of another heap object. See Resetting the Interpreter Environment in the chapter CINT the C++ Interpreter.
The subject of this chapter is CINT, ROOT's command line interpreter and script processor. First, we explain what CINT is and why ROOT uses it. Then CINT as the command line interpreter, the CINT commands, and CINT's extensions to C++ are discussed. CINT as the script interpreter is also explained and illustrated with several examples.

What is CINT?

CINT, which is pronounced C-int, is a C++ interpreter. An interpreter takes a program, in this case a C++ program, and carries it out by examining each instruction and in turn executing the equivalent sequence of machine language. For example, an interpreter translates and executes each statement in the body of a loop "n" times. It does not generate a machine language program. This may not be a good example, because most interpreters have become 'smart' about loop processing.

A compiler on the other hand, takes a program and makes a machine language executable. Once compiled the execution is very fast, which makes a compiler best suited for the case of "built once, run many times". For example, the ROOT executable is compiled occasionally and runs very fast. On the average, a compiled program runs ten times faster than an interpreted one.

Because it takes much time to compile, using a compiler is cumbersome for rapid prototyping when one changes and rebuilds as often as every few minutes. An interpreter, optimized for code that changes often and runs a few times, is the perfect tool for this.

Most of the time, an interpreter has a separate scripting language, such as Python, IDL, and PERL, designed especially for interpretation, rather than compilation. However, the advantage of having one language for both is that once the prototype is debugged and refined, it can be compiled without translating the code to a compiled language.

CINT being a C++ interpreter is the tool for rapid prototyping and scripting in C++. It is a stand-alone product developed by Masaharu Goto. It's executable comes with the standard distribution of ROOT ($ROOTSYS/bin/cint), and it can also be installed separately from:

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

This page also has links to all the CINT documentation. The downloadable tar file contains documentation, the CINT executable, and many demo scripts, which are not included in the regular ROOT distribution.

Here is a list of CINT's main features:

- Supports K&R-C, ANSI-C, and ANSI-C++
  CINT covers 80-90% of the K&R-C, ANSI-C and C++ language constructs. It supports multiple inheritance, virtual function, function overloading, operator overloading, default parameter, template, and much more. CINT is robust enough to interpret its own source code. CINT is not designed to be a 100% ANSI/ISO compliant C++ language processor. It is a portable scripting language environment, which is close enough to the standard C++.
- Interprets Large C/C++ source code
  CINT can handle huge C/C++ source code, and loads source files quickly. It can interpret its own, over 70,000 lines source code.
- Enables mixing Interpretation & Native Code
  Depending on the need for execution speed or the need for interaction, one can mix native code execution and interpretation. "makeCINT" encapsulates arbitrary C/C++ objects as a precompiled libraries. A precompiled library can be configured as a dynamically linked library. Accessing interpreted code and precompiled code can be done seamlessly in both directions.
- Provides a Single-Language solution
  CINT/makeCINT is a single-language environment. It works with any ANSI-C/C++ compiler to provide the interpreter environment on top of it.
- Simplifies C++
  CINT is meant to bring C++ to the non-software professional. C++ is simpler to use in the interpreter environment. It helps the non-software professional (the domain expert) to talk the same language as the software counterpart.
- Provides RTTI and a Command Line
  CINT can process C++ statements from command line, dynamically define/erase class definition and functions, load/unload source files and libraries. Extended Run Time Type Identification is provided, allowing you to explore unthinkable way of using C++.
- Has a Built-in Debugger and Class Browser
  CINT has a built-in debugger to debug complex C++ code. A text based class browser is part of the debugger.
- Is Portable
  CINT works on number of operating systems: HP-UX, Linux, SunOS, Solaris, AIX, Alpha-OSF, IRIX, FreeBSD, NetBSD, NEC EWS4800, NewsOS, BeBox, Windows-NT, Windows-9x, MS-DOS, MacOS, VMS, NextStep, Convex.
The ROOT Command Line Interface

Start up a ROOT session by typing ROOT at the system prompt.

```
$ root
```

```
*******************************************
*                                         *
*        W E L C O M E  to  R O O T       *
*                                         *
*   Version   2.25/02      21 August 2000 *
*                                         *
*  You are welcome to visit our Web site  *
*          http://root.cern.ch            *
*                                         *
*******************************************
```

Now create a `TLine` object:

```
root [] TLine l
root [] l.Print()
root [] l.SetX1(10)
root [] l.SetY1(11)
root [] l.Print()
```
Now let's execute a multi-line command:

```
root [] {
end with '}'
TLine l;
end with '}'
for (int i = 0; i < 5; i++) {
end with '}'
  l.SetX1(i);
end with '}'
  l.SetY1(i+1);
end with '}'
  l.Print();
end with '}'
}
```

Here we note:

- A multi-line command starts with a `{` and ends with a `}`.
- Every line has to be correctly terminated with a `;` (like in "real" C++)
- All objects are created in global scope.
- There is no way to back up, you are better off writing a script.
- Use `.q` to exit root.

---

The ROOT Script Processor

ROOT script 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.

**Un-named Scripts**

Let's start with a script containing a simple list of statements (like the multi-command line example given in the previous section). This type of script must start with a `{` and end with a `}` and is called an un-named script. Assume the file is called `script1.C`

```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 `script1.C` do:

```
root [] .x script1.C
```

This loads the contents of file `script1.C` and executes all statements in the interpreter's global scope.

---

One can re-execute the statements by re-issuing `.x script1.C` (since there is no function entry point).

Scripts are searched for in the `Root.MacroPath` as defined in your `.rootrc` file.

To check which script is being executed use:

```
root [] .which script1.C
/home/rdm/root/.script1.C
```

**Named Scripts**

Let's change the un-named script to a named script. Copy file `script1.C` to `script2.C` and add a function statement. Like this:

```c++
#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 `{` are required in this case. To execute function `main()` in `script2.C` do:

```
root [] .L script2.C // load script in memory
root [] main() // execute entry point main
Hello
x = 3 y = 5 i = 101
(int)0
root [] main() // execute main() again
Hello
x = 3 y = 5 i = 101
(int)0
root [] .func // list all functions known by CINT
filename line:size busy function type and name
...script2.C 4:9 0 public: int main();
```

The last command shows that `main()` has been loaded from file `script2.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 script2.C to script3.C and change the function name from main() to script3(int j = 10):

```c
#include <iostream.h>
int script3(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 script3() in script3.C type:

```
root [] .x script3.C(8)
```

This loads the contents of file script3.C and executes entry point script3(8).

Note that the above only works when the filename (minus extension) and function entry point are both the same. Function script3() can still be executed multiple times:

```
root [] script3()
Hello
x = 3 y = 5 i = 10
(int)0
root [] script3(33)
Hello
x = 3 y = 5 i = 33
(int)0
```

In a named script, the objects created on the stack are deleted when the function exits. For example, this scenario is very common. You create a histogram in a named script on the stack. You draw the histogram, but when the function exits the canvas is empty and the histogram disappeared.

To avoid histogram from disappearing you can create it on the heap (by using new). This will leave the histogram object intact, but the pointer in the named script scope will be deleted.

Since histograms (and trees) are added to the list of objects in the current directory, you can always retrieve them to delete them if needed.

```
root [] TH1F *h = (TH1F*)gDirectory->Get("myHist");
```

In addition, histograms and trees are automatically deleted when the current directory is closed. This will automatically take care of the clean up. See chapter Input/Output.

**Executing a Script from a Script**

You may want to execute a script conditionally inside another script. To do that you need to call the interpreter and you can do that with TROOT::ProcessLine().

Here is an example from \$ROOTSYS/tutorials/cernstaff.C that calls a script to build the root file if it does not exist:

```c
void cernstaff () {
    if (gSystem->AccessPathName("cernstaff.root")) {
        gROOT->ProcessLine(".x cernbuild.C");
    }
```

ProcessLine takes a parameter which is a pointer to an int or to a TInterpreter::EErrorCode to let you access the CINT error code after an attempt to interpret. This will contain the CINT error as defined in enum TInterpreter::EErrorCode.

**Resetting the Interpreter Environment**

Variables created on the command line and in un-named scripts are in the interpreter's global scope, which makes the variables created in un-named scripts available on the command line event after the script is done executing. This is the opposite of a named script where the stack variables are deleted when the function in which they are defined has finished execution.

When running an un-named script over again and this is frequently the case since un-named scripts are used to prototype, one should reset the global environment to clear the variables. This is done by calling gROOT->Reset(). It is good practice, and you will see this in the examples, to begin an un-named script with gROOT->Reset. It clears the global scope to the state just before executing the previous script (not including any logon scripts).

The gROOT->Reset() calls the destructor of the objects if the object was created on the stack. If the object was created on the heap (via new) it is not deleted, but the variable is no longer associated with it. Creating variables on the heap in un-named scripts and calling gROOT->Reset() without you calling the destructor explicitly will cause a memory leak.

This may be surprising, but it follows the scope rules. For example, creating an object on the heap in a function (in a named script) without explicitly deleting it will also cause a memory leak. Since when exiting the function only the stack variables are deleted.

The code below shows gROOT->Reset calling the destructor for the stack variable, but not for the heap variable. In the end, neither variable is available, but the memory for the heap variable is not released.

Here is an example.

```
root [] gDebug = 1
(const int)1
root [] TFile stackVar("stack.root","RECREATE")
<TKey Writing 86 bytes at address 64 for ID= stack.root Title=
root [] TFile *heapVar = new TFile("heap.root", "RECREATE")
<TKey Writing 84 bytes at address 64 for ID= heap.root Title=
```

We turn on Debug to see what the subsequent calls are doing. Then we create two variables, one on the stack and one on the heap.
When we call `gROOT->Reset()`, CINT tells us that the destructor is called for the stack variable, but it doesn't mention the heap variable.

Neither variable is available in after the call to reset.

The object on the stack is deleted and shows a null pointer when we do a `FindObject`. However, the heap object is still around and taking up memory.

---

### A Script Containing a Class Definition

Let's create a small class `TMyClass` and a derived class `TChild`. The virtual `TMyClass::Print()` method is overridden in `TChild`. Save this in file called `script4.C`.

```cpp
#include <iostream.h>

class TMyClass {
    private:
        float fX; // x position in centimeters
        float fY; // y position in centimeters
    public:
        TMyClass() { fX = fY = -1; }
        virtual void Print() const;
        void SetX(float x) { fX = x; }
        void SetY(float y) { fY = y; }
    };

    void TMyClass::Print() const // parent print method
    {
        cout << "fX = " << fX <<", fY = " << fY << endl;
    }

    //=======================================
    class TChild : public TMyClass {
        public:
            void Print() const;
    };

    void TChild::Print() const // child print metod
    {
        cout << "This is TChild::Print()\n" << endl;
        TMyClass::Print();
    }
```
To execute `script4.C` do:

```c
root [] .L script4.C
root [] TMyclass *a = new TChild
root [] a->Print()
This is TChild::Print()
fX = -1, fY = -1
root [] a->SetX(10)
root [] a->SetY(12)
root [] a->Print()
This is TChild::Print()
fX = 10, fY = 12
root [] .class TMyclass
```

As you can see an interpreted class behaves just like a compiled class.

There are some limitations for a class created in a script:

1. They cannot inherit from `TObject`. Currently the interpreter cannot patch the virtual table of compiled objects to reference interpreted objects.
2. Because the I/O is encapsulated in `TObject` and a class defined in a script can not inherit from `TObject`, it can not be written to a ROOT file.

For ways to add a class with a shared library and with ACLIC, see the chapter: "Adding a Class"

---

**Debugging Scripts**

A powerful feature of CINT is the ability to debug interpreted 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 `script4.C` still loaded, we can then do:

```c
root [] .b TChild::Print
Break point set to line 26 script4.C
root [] a.Print()

26   TChild::Print() const
27   {
28      cout << "This is TChild::Print()" << endl;
FILE:script4.C LINE:28 cint>
29      MyClass::Print();
FILE:script4.C LINE:29 cint>
30   }
31   }
```

As you can see an interpreted class behaves just like a compiled class.

There are some limitations for a class created in a script:

1. They cannot inherit from `TObject`. Currently the interpreter cannot patch the virtual table of compiled objects to reference interpreted objects.
2. Because the I/O is encapsulated in `TObject` and a class defined in a script can not inherit from `TObject`, it can not be written to a ROOT file.

For ways to add a class with a shared library and with ACLIC, see the chapter: "Adding a Class"
Inspecting Objects

An object of a class inheriting from TObject can be inspected, with the Inspect method. The TObject::Inspect method creates a window listing the current values of the objects members. For example, this is a picture of TFile.

You can see the pointers are in red and can be clicked on to follow the pointer to the object. For example, here we clicked on fKeys, the list of keys in memory.

ROOT/CINT Extensions to C++

In the next example, we demonstrate three of the most important extensions ROOT/CINT makes to C++. Start ROOT in the directory $ROOTSYS/tutorials (make sure to have first run .x hsimple.C):

You can see the pointers are in red and can be clicked on to follow the pointer to the object. For example, here we clicked on fKeys, the list of keys in memory.

The first command shows the first extension; the declaration of f may be omitted when "new" is used. CINT will correctly create f as pointer to 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 de-referencing syntax "->" but can use the simple "." notation.
The third extension is more important. In case CINT cannot find an object being referenced, it will ask ROOT to search for an object with an identical name in the search path defined by $ROOT::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 much typing. In this example, ROOT searches for hpx and finds it in simple.root.

The fourth is shown below. There is no need to put a semicolon at the end of a line. The difference between having it and leaving it off is that when you leave it off the return value of the command will be printed on the next line. For example:

\begin{verbatim}
root[] 23+5 // no semicolon prints the return value
\end{verbatim}

\begin{verbatim}
root[] 23+5; // semicolon no return value is printed
\end{verbatim}

Be aware that these extensions do not work when the interpreter is replaced by a compiler. Your code will not compile, hence when writing large scripts, it is best to stay away from these shortcuts. It will save you from having problems compiling your scripts using a real C++ compiler.

**ACLiC - The Automatic Compiler of Libraries for CINT**

Instead of having CINT interpret your script there is a way to have your scripts compiled, linked and dynamically loaded using the C++ compiler and linker. The advantage of this is that your scripts will run with the speed of compiled C++ and that you can use language constructs that are not fully supported by CINT. On the other hand, you cannot use any CINT shortcuts (see CINT extensions) and for small scripts, the overhead of the compile/link cycle might be larger than just executing the script in the interpreter.

ACLiC will build a CINT dictionary and a shared library from your C++ script, using the compiler and the compiler options that were used to compile the ROOT executable. You do not have to write a makefile remembering the correct compiler options, and you do not have to exit ROOT.

**Usage**

Before you can compile your interpreted script you need to add include statements for the classes used in the script. Once you did that, you can build and load a shared library containing your script. To load it, use the .L command and append the file name with a “++.”

\begin{verbatim}
root[] .L MyScript.C+
\end{verbatim}

The + option generates the shared library and naming it by taking the name of the file "filename" but replacing the dot before the extension by an underscore and by adding the shared library extension for the current platform.

For example on most platforms, hsimple.cxx will generate hsimple_cxx.so.

It uses the directive fMakeSharedLibs to create a shared library. If loading the shared library fails, it tries to output a list of missing symbols by creating an executable (on some platforms like OSF, this does not HAVE to be an executable) containing the script. It uses the directive fMakeExec to do so. For both directives, before passing them to TSystem::Exec, it expands the variables $ObjectFiles, $SharedLib, $LibName, $IncludePath, $LinkedLibs, $AppName and $ObjectFiles. See SetMakeSharedLib() for more information on those variables.

If we execute a .files command we can see the newly created shared library is in the list of loaded files.

The + command will check for a more recent timestamp on the script and the script header file before rebuilding the shared library. Note that it does not automatically check the time stamp of the include files except for the one that has the same name as the script with the header extension.

To ensure that the shared library is rebuilt you can use the ++ syntax:

\begin{verbatim}
root[] .L MyScript.C++
\end{verbatim}

To build, load, and execute the function with the same name as the file you can use the .x command. This is the same as executing a named script. You can have parameters and use .x or .X. The only difference is you need to append a + or a ++.

\begin{verbatim}
root[] .x MyScript.C+ (4000)
\end{verbatim}

Creating shared library
/home/./MyScript.C_so

The alternative to .L is to use gROOT::LoadMacro. For example, in one script you can use ACLiC to compile and load another script.

\begin{verbatim}
gROOT->LoadMacro("MyScript.C+")
gROOT->LoadMacro("MyScript.C++")
\end{verbatim}

* and ++ have the same meaning as described above. You can also use the gROOT::Macro method to load and execute the script.

\begin{verbatim}
gROOT->Macro("MyScript.C+")
\end{verbatim}

NOTE: You should not call ACLiC with a script that has a function called main(). When ACLiC calls rootcint with a function called main it tries to add every symbol it finds while parsing the script and the header files to the dictionary. This includes the system header files and the ROOT header files. This will result in duplicate entries at best and crashes at worst, because some classes in ROOT needs special attention before they can be added to the dictionary.

**Intermediate Steps and Files**

ACLiC executes two steps and a third one if needed. These are:

- Calling rootcint to create a CINT dictionary. rootcint is a ROOT specific version of makecint, CINT’s generic dictionary generator.
- Calling the compiler to build the shared library from the script.
- For more information on the use of files, see the documentation for $ObjectFiles, $SharedLib, $LibName, $IncludePath, $LinkedLibs, $AppName and $ObjectFiles. See SetMakeSharedLib() for more information on those variables.

The + command will check for a more recent timestamp on the script and the script header file before rebuilding the shared library. Note that it does not automatically check the time stamp of the include files except for the one that has the same name as the script with the header extension.

To ensure that the shared library is rebuilt you can use the ++ syntax:

\begin{verbatim}
root[] .L MyScript.C++
\end{verbatim}

To build, load, and execute the function with the same name as the file you can use the .x command. This is the same as executing a named script. You can have parameters and use .x or .X. The only difference is you need to append a + or a ++.

\begin{verbatim}
root[] .x MyScript.C+ (4000)
\end{verbatim}

Creating shared library
/home/./MyScript.C_so

The alternative to .L is to use gROOT::LoadMacro. For example, in one script you can use ACLiC to compile and load another script.

\begin{verbatim}
gROOT->LoadMacro("MyScript.C+")
gROOT->LoadMacro("MyScript.C++")
\end{verbatim}

* and ++ have the same meaning as described above. You can also use the gROOT::Macro method to load and execute the script.

\begin{verbatim}
gROOT->Macro("MyScript.C+")
\end{verbatim}

NOTE: You should not call ACLiC with a script that has a function called main(). When ACLiC calls rootcint with a function called main it tries to add every symbol it finds while parsing the script and the header files to the dictionary. This includes the system header files and the ROOT header files. This will result in duplicate entries at best and crashes at worst, because some classes in ROOT needs special attention before they can be added to the dictionary.

**Intermediate Steps and Files**

ACLiC executes two steps and a third one if needed. These are:

- Calling rootcint to create a CINT dictionary. rootcint is a ROOT specific version of makecint, CINT’s generic dictionary generator.
- Calling the compiler to build the shared library from the script.
These types of headers can be included in interpreted and compiled mode:

In a script you can set the include path:

```
#include "-I$HOME/mypackage/include 
```

You can append to the include path by typing:

```
#include "-I$HOME/mypackage/include 
```

You can get the include path by typing:

```
#include " -I 
```

Setting the Include Path

You can get the include path by typing:

```
#include " -I 
```

In a script you can set the include path:

```
#include " -I 
```

Moving between Interpreter and Compiler

The best way to develop portable scripts is to make sure you can always run them with both, the interpreter and with ACLiC. To do so, do not use the C INT extensions and program around the C INT limitations. When it is not possible or desirable to program around the CINT limitations, you can use the C preprocessor symbols defined for CINT and rootcint.

The preprocessor symbol __CINT__ is defined for both CINT and rootcint. The symbol __MAKECINT__ is only defined in rootcint.

Use `#ifdef(__CINT__)` || `#ifdef(__MAKECINT__)` to bracket code that needs to be seen by the compiler and rootcint, but will be invisible to the interpreter.

For example, the following will hide the declaration and initialization of the array `gArray` from both CINT and rootcint.

```
#define __CINT__
int gArray[] = { 2, 3, 4};
#endif
```

Because ACLIC calls rootcint to build a dictionary, the declaration of `gArray` will not be included in the dictionary, and consequently, `gArray` will not be available at the command line even if ACLIC is used. CINT and rootcint will ignore all statements between the `#if defined(__CINT__)` and `#endif`. If you want to use `gArray` in the same script as its declaration, you can do so. However, if you want the script in the interpreter you have to bracket the usage of `gArray` between `#if's`, since the definition is not visible.

If you add the following preprocessor statements, `gArray` will be visible to rootcint but still not visible to CINT. If you use ACLIC, `gArray` will be available at the command line and be initialized properly by the compiled code.

```
#define __CINT__
int gArray[] = { 2, 3, 4};
#elif defined(__MAKECINT__)
int gArray[];
#endif
```

We recommend you always write scripts with the needed include statements. In most cases, the script will still run with the interpreter. However, a few header files are not handled very well by CINT.

These types of headers can be included in interpreted and compiled mode:

- The subset of standard C/C++ headers defined in `$ROOTSYS/cint/include`.
- Headers of classes defined in a previously loaded library (including ROOT's own). The defined class must have a name known to ROOT (i.e. a class with a `ClassDef`).

A few headers will cause problems when they are included in interpreter mode, because they are already included by the interpreter itself. In general, the interpreter needs to know whether to use the interpreted or compiled version. The mode of the definition needs to match the mode of the reference.

Here are the cases that need to be excluded in interpreted mode, but included for rootcint. Bracket these with:

```
#ifdef(__CINT__) || defined(__MAKECINT__) 
```

- All CINT headers, see `$ROOTSYS/cint/inc`.
- Headers with classes named other than the file name. For example:
  ```
  Rtypes.h and GuiTypes.h.
  ```
- Headers with a class defined in a libraries before the library is loaded. For example: having a `#include "TLorenzVector.h"` before `#include "libPhysics.h"`
  ```
  This will also cause problems when compiling the script, but a clear error message will be given. With the interpreter it may core dump. Bracket these type of include statements with `#if defined(__CINT__)`, this will print an error in both modes.

Hiding header files from rootcint that are necessary for the compiler but optional for the interpreter can lead to a subtle but fatal errors. For example:

```
#define _CINT_ 
#include "TTree.h"
```

```
Rootcint: include "TTree.h"
```

In this case, rootcint does not have enough information about the `TTree` class to produce the correct dictionary file. If you try this, rootcint and compiling will be error free, however, instantiating a `TTree` object from the CINT command line will cause a fatal error.

In general it is recommended to let rootcint see as many header files as possible.

```
#include "TTree.h"
```

In a future release, we plan to add combine scripts from different files into one library by using `#include` statements;
gSystem->SetIncludePath("-I$HOME/mypackage/include")

The $ROOTSYS/include directory is automatically appended to the include path, so you don't have to worry about including it, however if you have already added a path, this command will overwrite it.
Object Ownership

An object has ownership of another object if it has permission to delete it. Usually ownership is held by a collection or a parent object such as a pad.

To prevent memory leaks and multiple attempts to delete an object, you need to know which objects are owned by ROOT and which are owned by you.

The following rules apply to the ROOT classes.

- Histograms, trees, and event lists created by the user are owned by current directory (gDirectory). When the current directory is closed or deleted the objects it owns are deleted also.

- The TROOT master object (gROOT) has several collections of objects. Objects that are members of these collections are owned by gROOT (see the paragraph “Ownership by the Master TROOT Object (gROOT)” below).

- Objects created by another object, for example the function object (e.g. TF1) created by the TH1::Fit method is owned by the histogram.

- An object created by DrawCopy methods, is owned by the pad it is drawn in.

If an object fits none of these cases, the user has ownership. The next paragraphs describe each rule and user ownership in more detail.

Ownership by Current Directory (gDirectory)

When a histogram, tree, or event list (TEventList) is created, it is added to the list of objects in the current directory by default. You can get the list of objects in a directory and retrieve a pointer to a specific object with the GetList method. This example retrieves a histogram.

```c
TH1F *h = (TH1F*)gDirectory->GetList()->FindObject("myHist");
```

The method `TDirectory::GetList()` returns a TList of objects in the directory. It looks in memory, and is implemented in all ROOT collections.

You can change the directory of a histogram, tree, or event list with the SetDirectory method. Here we use a histogram for an example, but the same applies to trees and event lists.

```c
h->SetDirectory(newDir)
```

You can also remove a histogram from a directory by using `SetDirectory(0)`. Once a histogram is removed from the directory, it will not be deleted when the directory is closed. It is now your responsibility to delete this histogram once you have finished with it.

To change the default that automatically adds the histogram to the current directory, you can call the static function:

```c
TH1::AddDirectory(kFALSE);
```

All histograms created hereafter will not be added to the current directory. In this case, you own all histogram objects and you will need to delete them and clean up the references.

You can still set the directory of a histogram by calling `SetDirectory` once it has been created as described above.

Note that, when a file goes out of scope or is closed all objects on its object list are deleted.

Ownership by the Master TROOT Object (gROOT)

The master object gROOT, maintains several collections of objects. For example, a canvas is added to the collection of canvases and it is owned by the canvas collection.

```c
TSeqCollection* fFiles        List of files (TFile)
TSeqCollection* fMappedFiles   List of memory mapped files (TMappedFile)
TSeqCollection* fSockets       List of network sockets (TSocket and TServerSocket)
TSeqCollection* fCanvases      List of canvases (TCanvas)
TSeqCollection* fStyles        List of styles (TStyle)
TSeqCollection* fFunctions     List of analytic functions (TF1, TF2, TF3)
TSeqCollection* fTasks         List of tasks (TTask)
TSeqCollection* fColors        List of colors (TColor)
TSeqCollection* fSpecials      List of special objects
TSeqCollection* fCleanups      List of recursiveRemove collections
```

These collections are also displayed in the root folder of the Object Browser.

Most of these collections are self explanatory. The special cases are the collections of specials and cleanups.

The Collection of Specials

This collection contains objects of the following classes: TCut, TMultiDimFit, TPrincipal, TChains. In addition it contains the gHtml object, gMinuit objects, and gLite objects.
the array of contours graphs (TGraph) created when calling the Draw method of a histogram with the "CONT, LIST" option.

Access to the Collection Contents

The current content for the collection listed above can be accessed with the corresponding gROOT->GetListOf method (for example gROOT->GetListOfCanvases). In addition, gROOT->GetListOfBrowsables returns a collection of all objects visible on the left side panel in the browser (see the image of the Object Browser above).

Ownership by Other Objects

When an object is created by another, the creating object is the owner of the one it created. For example:

```c
myHisto->Fit("gaus")
```

The call to Fit copies the global TF1 object gaus and attaches the copy to the histogram. When the histogram is deleted, the copy of gaus is deleted also.

When a pad is deleted or cleared, all objects in the pad with the kCanDelete bit set are automatically deleted. Currently the objects created by the DrawCopy methods, have the kCanDelete bit set and are therefore owned by the pad.

Ownership by the User

The user owns all objects not described in one of the above cases.

TObject has two bits, kCanDelete and kMustCleanup, that influence how an object is managed (in TObject::fBits). These are in an enumeration in TObject.h. To set these bits do:

```c
MyObject->SetBit(kCanDelete)
MyObject->SetBit(kMustCleanup)
```

The bits can be reset and tested with the TObject::ResetBit and TObject::TestBit methods.

The kCanDelete Bit

The gROOT collections (see above) own their members and will delete them regardless of the kCanDelete bit. In all other collections, when the collection Clear method is called (i.e. TList::Clear()), members with the kCanDelete bit set are deleted and removed from the collection. If the kCanDelete bit is not set, the object is only removed from the collection but not deleted.

If a collection Delete(TList::Delete()) method is called, all objects in the collection are deleted without considering the kCanDelete bit.

It is important to realize that deleting the collection ([i.e. delete MyCollection]), DOES NOT delete the members of the collection. If the user specified MyCollection->SetOwner() the collection owns the objects and delete MyCollection will delete all its members. Otherwise you need to:

```c
// delete all member objects in the collection
MyCollection->Delete();
// and delete the collection object
delete MyCollection;
```

Note that kCanDelete is automatically set by the DrawCopy method and it can be set for any object by the user.

For example, all graphics primitives must be managed by the user. If you want TCanvas to delete the primitive you created you have to set the kCanDelete bit.

The kCanDelete bit setting is displayed with TObject::Is(). The last number is either 1 or 0 and is the kCanDelete bit.

The kMustCleanup Bit

When the kMustCleanup bit is set, the object destructor will remove the object and its references from all collections in the clean up collection (gROOT::fCleanups).

An object can be in several collections, for example if an object is in a browser and on two canvases. If the kMustCleanup bit is set, it will automatically be removed from the browser and both canvases when the destructor of the object is called.

kMustCleanup is set

- When an object is added to a pad (or canvas) in TObject::AppendPad.
- When an object is added to a TBrowser with TBrowser::Add.
- When an object is added to a TFolder with TFolder::Add.
• When creating an inspector canvas with TInspectCanvas::Inspector.
• When creating a TCanvas.
• When painting a frame for a pad, the frame’s kMustClean up is set in TPad::PaintPadFrame.

The user can add his own collection to the collection of clean ups, to take advantage of the automatic garbage collection. For example:

```c
// create two list
TList *myList1, *myList2;
// add both to of clean up
gROOT->GetListOfCleanUps()->Add(myList1);
gROOT->GetListOfCleanUps()->Add(myList2);
// assuming myObject is in myList1 and myList2, when calling:
delete myObject;
// the object is deleted from both lists
```
9 Graphics and the Graphical User Interface

Graphical capabilities of ROOT range from 2D objects (lines, polygons, arrows) to various plots, histograms, and 3D graphical objects. In this chapter, we are going to focus on principals of graphics and 2D objects. Plots and histograms are discussed in a chapter of their own.

Drawing Objects

In ROOT, most objects derive from a base class TObject. This class has a virtual method Draw() so all objects are supposed to be able to be "drawn". The basic whiteboard on which an object is drawn is called a canvas (defined by the class TCanvas). If several canvases are defined, there is only one active at a time. One draws an object in the active canvas by using the statement:

```
object.Draw()
```

This instructs the object "object" to draw itself. If no canvas is opened, a default one (named "c1") is instantiated and drawn. Try the following commands:

```
root [] TLine a (0.1,0.1,0.6,0.6)
root [] a.Draw()
```

The first statement defines a line and the second one draws it. A default canvas is drawn since there was no opened one.

Interacting with Graphical Objects

When an object is drawn, one can interact with it. For example, the line drawn in the previous paragraph may be moved or transformed. One very important characteristic of ROOT is that transforming an object on the screen will also transform it in memory. One actually interacts with the real object, not with a copy of it on the screen. You can try for instance to look at the starting X coordinate of the line:

```
root [] a.GetX1()
```

You do not obtain the same result as before, the coordinates of 'a' have changed. As said, interacting with an object on the screen changes the object in memory.

Moving, Resizing and Modifying Objects

Changing the graphic objects attributes can be done with the GUI or programmatically. First, let's see how it is done in the GUI.

The Left Mouse Button

As was just seen moving or resizing an object is done with the left mouse button. The cursor changes its shape to indicate what may be done:

- Moving:
- Resizing:
- Rotate:
- Resize (exists also for the other directions):　
- Enlarge (used for text):
- Move:

Here are some examples of Moving:　Resizing:

Rotating:
With C++ Statements (Programmatically)

How would one move an object in a script? Since there is a tight correspondence between what is seen on the screen and the object in memory, changing the object changes it on the screen.

For example, try to do:

```c
root[1] a.SetX1(0.9)
```

This should change one of the coordinates of our line, but nothing happens on the screen. Why is that? In short, the canvas is not updated with each change for performance reasons. See the sub section on: "Updating the Pad" in the next section.

Selecting Objects

The Middle Mouse Button

Objects in a canvas, as well as in a pad, are stacked on top of each other in the order they were drawn. Some objects may become "active" objects, which means they are reordered to be on top of the others. To interactively make an object "active", you can use the middle mouse button. In case of canvases or pads, the border becomes highlighted when it is active.

With C++ Statements (Programmatically)

Frequently we want to draw in different canvases or pads. By default, the objects are drawn in the active canvas. To activate a canvas you can use the "TCanvas::cd()" method.

```c
root[1] c1->cd()
```

Context Menus: the Right Mouse Button

The context menus are a way to interactively call certain methods of an object. When designing a class, the programmer can add methods to the context menu of the object by making minor changes to the header file.

Using Context Menus

On a ROOT canvas, you can right-click on any object and see the context menu for it. The script `hSimple.c` draws a histogram. The image below shows the context menus for some of the objects on the canvas.

This picture shows that drawing a simple histogram involves as many as seven objects.

When selecting a method from the context menu and that method has options, the user will be asked for numerical values or strings to fill in the option. For example, `TAxis::SetTitle` will prompt you for a string to use for the axis title.

Structure of the Context Menus

The curious reader will have noticed that each entry in the context menu corresponds to a method of the class.

Look for example to the menu named `TAxis::xaxis`. `xaxis` is the name of the object and `TAxis` the name of its class. If we look at the list of `TAxis` methods, for example in [http://www.root.c/html/TAxis.html](http://www.root.c/html/TAxis.html), we see the methods `SetTimeDisplay` and `UnZoom`, which appear also in the context menu.
There are several divisions in the context menu, separated by lines. The top division is a list of the class methods; the second division is a list of the parent class methods. The subsequent divisions are the methods of other parent classes in case of multiple inheritance.

For example, see the TPaveText::title context menu. A TPaveText inherits from TAttLine, which has the method SetLineAttributes().

Adding Context Menus for a Class

For a method to appear in the context menu of the object it has to be marked by // *MENU* in the header file. Below is the line from TAttLine.h that adds the SetLineAttributes method to the context menu.

```cpp
virtual void SetLineAttributes(); // *MENU*
```

Nothing else is needed, since CINT knows the classes and their methods. It takes advantage of that to create the context menu on the fly when the object is clicking on.

If you click on an axis, ROOT will ask the interpreter what are the methods of the TAxis and which ones are set for being displayed in a context menu.

Now, how does the interpreter know this? Remember, when you build a class that you want to use in the ROOT environment, you use rootcint that builds the so-called stub functions and the dictionary. These functions and the dictionary contain the knowledge of the used classes. To do this, rootcint parses all the header files.

ROOT has defined some special syntax to inform CINT of certain things, this is done in the comments so that the code still compiles with a C++ compiler.

For example, you have a class with a Draw() method, which will display itself. You would like a context menu to appear when on clicks on the image of an object of this class. The recipe is the following:

1. The class has to contain the ClassDef/ClassImp macros
2. For each method you want to appear in the context menu, put a comment after the declaration containing *MENU* or *TOGGLE* depending on the behavior you expect. One usually uses Set methods (setters).

For example:

```cpp
class MyClass : public TObject
private:
  int fV1; // first variable
double fV2; // second variable
public:
  int GetV1() {return fV1;} // *MENU*
  void SetV1(int x1) {fV1 = x1;} // *MENU*
  void SetV2(double d2) {fV2 = d2;} // *MENU*
  void SetMyBool(int x1, double d2) {fV1 = x1; fV2 = d2;}
ClassDef(MyClass,1)
```

The *TOGGLE* comment is used to toggle a boolean data field. In that case, it is safe to call the data field MyBool where MyBool is the name of the setter SetMyBool. Replace MyBool with your own boolean variable.

3. You can specify arguments and the data members in which to store the arguments.

For example:

```cpp
void SetXXX(Int_t x1, Float_t y2); // *MENU* *ARGS={x1=>fV1}
```

This statement is in the comment field, after the *MENU*. If there is more than one argument, these arguments are separated by commas, where fV1 and fV2 are data fields in the same class.

```cpp
void SetXXX(Int_t x1, Float_t y2); // *MENU* *ARGS={x1=>fX1,y2=>fY2}
```

If the arguments statement is present, the option dialog displayed when selecting SetXXXfield will show the values of variables. We indicate to the system which argument corresponds to which data member of the class.

Executing Events when a Cursor passes on top of an Object

This paragraph is for class designers. When a class is designed, it is often desirable to include drawing methods for it. We will have a more extensive discussion about this, but drawing an object in a canvas or a pad consists in "attaching" the object to that pad. When one uses object.Draw(), the object is NOT painted at this moment. It is only attached to the active pad or canvas.

Another method should be provided for the object to be painted, the Paint() method. This is all explained in the next paragraph.

As well as Draw() and Paint(), other methods may be provided by the designer of the class. When the mouse is moved or a button pressed/released, the TCanvas function named HandleInput() scans the list of objects in all its pads and for each object calls some standard methods to make the object react to the event (mouse movement, click or whatever).

The second one is DistanceToPrimitive(px,py). This function computes a "distance" to an object from the mouse position at the pixel position (px,py, see definition at the end of this paragraph) and returns this distance in pixel units. The selected object will be the one with the shortest computed distance. To see how this works, 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, a histogram, the status line indicates the name of the histogram, the position x,y in histogram coordinates, the channel number and the channel content. It's nice for the canvas to know what is the closest object from the mouse, but it's even nicer to be able to make this object react. The third standard method to be provided is ExecuteEvent(). This method actually does the event reaction.

Its prototype is where px and py are the coordinates at which the event occurred, except if the event is a key press, in which case px contains the key code.
void ExecuteEvent(Int_t event, Int_t px, Int_t py);

Where event is the event that occurs and is one of the following (defined in Buttons.h):

kNoEvent, kButton1Down, kButton2Down, kButton3Down,
kButton1Up, kButton2Up, kButton3Up, kButton1Motion,
kButton2Motion, kButton1Locate, kButton2Locate, kButton1Double,
kButton2Double, kButton3Double, kKeyDown, kKeyUp,
sKeyPress, kMouseMotion, kMouseEnter, kMouseLeave.

We hope the names are self-explanatory.

Designing an ExecuteEvent method is not very easy, except if one wants very basic treatment. We will not go into that and let the reader refer to the sources of classes like TLine or TBox. Go and look at their ExecuteEvent method!

We can nevertheless give some reference to the various actions that may be performed. For example, one often wants to change the shape of the cursor when passing on top of an object. This is done with the SetCursor method:

```cpp
gPad->SetCursor(cursor)
```

The argument cursor is the type of cursor. It may be:

kBottomLeft, kBottomRight, kTopLeft, kTopRight,
kBottomSide, kLeftSide, kTopSide, kRightSide, kMove,
kCross, kArrowHor, kArrowVer, kHand, kRotate, kPointer,
kArrowRight, kCaret, kWatch.

They are defined in TVirtualX.h and again we hope the names are self-explanatory. If not, try them by designing a small class. It may derive from something already known like TLine.

Note that the ExecuteEvent() functions may in turn; invoke such functions for other objects, 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) use or redefine the picking functions in its base class.

The last comment is that mouse position is always given in pixel units in all these standard functions. px=0 and py=0 corresponds to the top-left corner of the canvas. Here, we have followed the standard convention in windowing systems. Note that user coordinates in a canvas (pad) have the origin at the bottom-left corner of the canvas (pad). This is all explained in the paragraph "Coordinate system of a pad".

---

**Graphical Containers: Canvas and Pad**

We have talked a lot about canvases, which may be seen as windows. More generally, a graphical entity that contains graphical objects is called a Pad. A Canvas is a special kind of Pad. From now on, when we say something about pads, this also applies to canvases.

A pad (class TPad) is a graphical container in the sense it contains other graphical objects like histograms and arrows. It may contain other pads (subpads) as well. More technically, each pad has a linked list of pointers to the objects it holds.

```cpp
Pad1       Arrow
          Text
          Subpad
          Histogram
          Label
          Polyline
```

Drawing an object is nothing more than adding its pointer to this list. Look for example at the code of TH1::Draw(). It is merely ten lines of code. The last statement is AppendPad(). This statement calls a method of TObject that just adds the pointer of the object, here a histogram, to the list of objects attached to the current pad. Since this is a TObject method, every object may be "drawn", which means attached to a pad.

We can illustrate this by the following figure.

The image correspond to this structure:
When is the painting done then?  The answer is: when needed. Every object that derives from TObject has a Paint() method. It may be empty, but for graphical objects, this routine contains all the instructions to effectively paint it in the active pad. Since a Pad has the list of objects it owns, it will call successively the Paint() method of each object, thus re-painting the whole pad on the screen. If the object is a sub-pad, its Paint() method will call the Paint() method of the objects attached, recursively calling paint() for all the objects.

The Global Pad: gPad

When an object is drawn, it is always in the so-called active pad. For every day use, it is comfortable to be able to access the active pad, whatever it is. For that purpose, there is a global pointer, called gPad. It is always pointing to the active pad. If you want to change the fill color of the active pad to blue but you don't know its name, do this.

root[] gPad->SetFillColor(38)

To get the list of colors, go to the paragraph "Color and color palettes" or if you have opened a canvas, click on the View menu, selecting the Colors item.

Finding an Object in a Pad

Now that we have a pointer to the active pad, gPad and that we know this pad contains some objects, it is sometimes interesting to access one of those objects. The method GetPrimitive() of TPad, i.e. TPad::GetPrimitive(const char* name) does exactly this. Since most of the objects that a pad contains derive from TObject, they have a name. The following statement will return a pointer to the object myobjectname and put that pointer into the variable obj. As you see, the type of returned pointer is (TObject*).

root[] obj = gPad->GetPrimitive("myobjectname")
<class TObject*>(0x1063cba8)

Even if your object is something more complicated, like a histogram TH1F, this is normal. A function cannot return more than one type. So the one chosen was the lowest common denominator to all possible classes, the class from which everything derives, TObject.

How do we get the right pointer then?

Simply do a cast of the function output that is transforming this output (pointer) into the right type. For example if the object is a TPaveLabel:

root[] obj = (TPaveLabel*)(gPad->GetPrimitive("myobjectname"))
<class TPaveLabel*>(0x1063cba8)

This works for all objects deriving from TObject. However, a question remains. An object has a name if it derives from TNamed, not from TObject. For example, an arrow (TArrow) doesn't have a name. In that case, the "name" is the name of the class. To know the name of an object, just click with the right button on it. The name appears at the top of the context menu.

In case of multiple unnamed objects, a call to GetPrimitive("className") returns the instance of the class that was first created. To retrieve a later instance you can use

GetListofPrimitives(), which returns a list of all the objects on the pad. From the list you can select the object you need.

Hiding an Object

Hiding an object in a pad can be made by removing it from the list of objects owned by that pad. This list is accessible by the GetListofPrimitives() method of TPad. This method returns a pointer to a TList. Suppose we get the pointer to the object, we want to hide, call it obj (see paragraph above). We get the pointer to the list:

root[] li = gPad->GetListofPrimitives()

Then remove the object from this list:

root[] li->Remove(obj)

The object will disappear from the pad as soon as the pad is updated (try to resize it for example).

If one wants to make the object reappear:

root[] obj->Draw()

Caution, this will not work with composed objects, for example many histograms drawn on the same plot (with the option "same"). There are other ways! Try to use the method described here for simple objects.

The Coordinate Systems of a Pad

Three coordinate systems may be used in a TPad: pixel coordinates, normalized coordinates (NDC), and user coordinates.

The User Coordinate System

The most common is the user coordinate system. Most methods of TPad use the user coordinates, and all graphic primitives have their parameters defined in terms of user coordinates. By default, when an empty pad is drawn, the user coordinates are set to a range from 0 to 1 starting at the lower left corner. At this point they are equivalent of the NDC coordinates (see below). If you draw a high level graphical object, such as a histogram or a function, the user coordinates are set to the coordinates of the histogram. Therefore, when you set a point it will be in the histogram coordinates.
For a newly created blank pad, one may use TPad::Range to set the user coordinate system. This function is defined as:

```cpp
void Range(float x1, float y1, float x2, float y2)
```

The arguments `x1, x2` defines the new range in the x direction, and the `y1, y2` define the new range in the y-direction.

```cpp
root[1] TCanvas MyCanvas("MyCanvas")
root[1] gPad->Range(-100, -100, 100, 100)
```

This will set the active pad to have both coordinates to go from -100 to 100, with the center of the pad at (0,0). You can visually check the coordinates by viewing the status bar in the canvas. To display the status bar select Options:Event Status in the canvas menu.

### The Normalized Coordinate System (NDC)

Normalized coordinates are independent of the window size and of the user system. The coordinates range from 0 to 1 and (0,0) correspond to the bottom-left corner of the pad. Several internal ROOT functions use the NDC system (3D primitives, PostScript, log scale mapping to linear scale). You may want to use this system if the user coordinates are not known ahead of time.

### The Pixel Coordinate System

The least common is the pixel coordinate system, used by functions such as DistanceToPrimitive() and ExecuteEvent(). Its primary use is for cursor position, which is always given in pixel coordinates. If (px, py) is the cursor position, px=0 and py=0 corresponds to the top-left corner of the pad, which is the standard convention in windowing systems.

### Using NDC for a particular Object

Most of the time, you will be using the user coordinate system. But sometimes, you will want to use NDC. For example, if you want to draw text always at the same place over a histogram, no matter what the histogram coordinates are. There are two ways to do this. You can set the NDC for an object or may convert NDC to user coordinates. Most graphical objects offer an option to be drawn in NDC. For instance, a line (TLine) may be drawn in NDC by using DrawLineNDC(). A latex formula or a text may use TText::SetNDC() to be drawn in NDC coordinates.

### Converting between Coordinates Systems

There are a few utility functions in TPad to convert from one system of coordinates to another. In the following table, a point is defined by (px, py) in pixel coordinates; (ux, uy) in user coordinates, (ndcx, ndcy) in NDC coordinates.

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Methods of TPad</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pixel to User</td>
<td>PixeltoX(px)</td>
<td>double</td>
</tr>
<tr>
<td></td>
<td>PixeltoY(py)</td>
<td>double</td>
</tr>
<tr>
<td></td>
<td>PixeltoXY(px, py, 4ux, 4uy)</td>
<td>changes ux, uy</td>
</tr>
<tr>
<td>NDC to Pixel</td>
<td>UtoPixel(ndcx)</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>VtoPixel(ndcy)</td>
<td>int</td>
</tr>
<tr>
<td>User to Pixel</td>
<td>XtoPixel(ux)</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>YtoPixel(uy)</td>
<td>int</td>
</tr>
<tr>
<td></td>
<td>XtoPixel(ux, uy, 4px, 4py)</td>
<td>changes px, py</td>
</tr>
</tbody>
</table>

### Dividing a Pad into Sub-pads

Dividing a pad into sub-pads in order for instance to draw a few histograms, may be done in two ways. The first is to build pad objects and to draw them into a parent pad, which may be a canvas. The second is to automatically divide a pad into horizontal and vertical sub pads.

#### Creating a Single Sub-pad

The simplest way to divide a pad is to build sub-pads in it. However, this forces the user to explicitly indicate the size and position of those sub-pads. Suppose we want to build a sub-pad in the active pad (pointed by `gPad`). First, we build it, using a TPad constructor:

```cpp
root[] subpad1 = new TPad("subpad1","The first subpad",.1,.1,.5,.5)
```

One gives the coordinates of the lower left point (0.1,0.1) and of the upper right one (0.5,0.5). These coordinates are in NDC. This means that they are independent of the user coordinates system, in particular if you have already drawn for example a histogram in the mother pad. The only thing left is to draw the pad:

```cpp
root[] subpad1->Draw()
```

If you want more sub-pads, you have to repeat this procedure as many times as necessary.
Dividing a Canvas into Sub-Pads

The manual way of dividing a pad into sub-pads is sometimes very tedious. There is a way to automatically generate horizontal and vertical sub-pads inside a given pad.

```root
pad1->Divide(3,2)
```

If `pad1` is a pad then, it will divide the pad into 3 columns of 2 sub-pads:

The generated sub-pads get names `pad1_i` where `i` is from 1 to `nm`. In our case `pad1_1, pad1_2...pad1_6`:

The names `pad1_1` etc... correspond to new variables in CINT, so you may use them as soon as the `pad->Divide()` was executed. However, in a compiled program, one has to access these objects. Remember that a pad contains other objects and that these objects may, themselves be pads. To do so you can force the painting with a `TCanvas::Update()`. For example if you want to access an object created during the paint process. To do so you can force the painting with a `TCanvas::Update()`. If you the internal histogram obtained with `TH1::GetHistogram()` is created only after the pad is painted. The pad is painted automatically after the script is finished executing or if you force the painting with `TPad::Modified` followed by a `TCanvas::Update()`. Note that it is not necessary to call `TPad::Modified` after a call to `Draw()`. The "bit-modified" is set automatically by `Draw()`.

A subsequent call to `TCanvas::Update()` scans the list of sub-pads and repaints the pads declared modified.

In compiled code or in a long macro, you may want to access an object created during the paint process. To do so you can force the painting with a `TCanvas::Update()`. For example if you want to access an object created during the paint process. To do so you can force the painting with a `TCanvas::Update()`. If you use `TPad::Modified` followed by a `TCanvas::Update()`, the sub pad has not been declared modified and it will not be updated.

Also note that a call to `pad->Update()` where `pad` is a sub pad of `canvas`, calls `canvas->Update()` and recursively updates all the pads on the canvas.

Updating the Pad

For performance reasons, a pad is not updated with every change. For example, changing the coordinates of the pad does not automatically redraw it. Instead, the pad has a "bit-modified" that triggers a redraw. This bit is automatically set by:

1. Touching the pad with the mouse. For example resizing it with the mouse.
2. Finishing the execution of a script.
3. Adding a new primitive or modifying some primitives for example the name and title of an object.

You can also set the "bit-modified" explicitly with the `Modified` method:

```c
// this pad has changed
root[] pad1->Modified()
// recursively update all modified pads:
root[] c1->Update()
```

A subsequent call to `TCanvas::Update()` scans the list of sub-pads and repaints the pads declared modified.

In compiled code or in a long macro, you may want to access an object created during the paint process. To do so you can force the painting with a `TCanvas::Update()`. For example if you want to access an object created during the paint process. To do so you can force the painting with a `TCanvas::Update()`. If you use `TPad::Modified` followed by a `TCanvas::Update()`. Note that it is not necessary to call `TPad::Modified` after a call to `Draw()`. The "bit-modified" is set automatically by `Draw()`.

A note about the "bit-modified" in sub pads: when you want to update a sub pad in your canvas, you need to call `pad->Modified()` rather than `canvas->Modified()`. This is done by adding two parameters to `Divide()`, which are the margins in x and y:

```c
// this pad has changed
root[] pad1->Modified()
// recursively update all modified pads:
root[] c1->Update()
```

Making a Pad Transparent

As we will see in the paragraph "Fill attributes", a fill style (type of hatching) may be set for a pad.

```c
root[] pad1->SetFillStyle(istyle)
```

This is done with the `SetFillStyle` method where `istyle` is a style number, defined in "Fill attributes".

A special set of styles allows handling of various levels of transparency. These are styles number 4000 to 4100, 4000 being fully transparent and 4100 fully opaque.

So, suppose you have an existing canvas with several pads. You create a new pad (transparent) covering for example the entire canvas. Then you can use the `Modified` method:

```c
// this pad has changed
root[] pad1->Modified()
// recursively update all modified pads:
root[] c1->Update()
```

A subsequent call to `TCanvas::Update()` scans the list of sub-pads and repaints the pads declared modified.

In compiled code or in a long macro, you may want to access an object created during the paint process. To do so you can force the painting with a `TCanvas::Update()`. For example if you want to access an object created during the paint process. To do so you can force the painting with a `TCanvas::Update()`. If you use `TPad::Modified` followed by a `TCanvas::Update()`. Note that it is not necessary to call `TPad::Modified` after a call to `Draw()`. The "bit-modified" is set automatically by `Draw()`.

A note about the "bit-modified" in sub pads: when you want to update a sub pad in your canvas, you need to call `pad->Modified()` rather than `canvas->Modified()`. This is done by adding two parameters to `Divide()`, which are the margins in x and y:

```c
// this pad has changed
root[] pad1->Modified()
// recursively update all modified pads:
root[] c1->Update()
```

A subsequent call to `TCanvas::Update()` scans the list of sub-pads and repaints the pads declared modified.

In compiled code or in a long macro, you may want to access an object created during the paint process. To do so you can force the painting with a `TCanvas::Update()`. For example if you want to access an object created during the paint process. To do so you can force the painting with a `TCanvas::Update()`. If you use `TPad::Modified` followed by a `TCanvas::Update()`, the sub pad has not been declared modified and it will not be updated.

Also note that a call to `pad->Update()` where `pad` is a sub pad of `canvas`, calls `canvas->Update()` and recursively updates all the pads on the canvas.
For example:

```c
root[] .x tutorials/h1draw.C
root[] TPad *newpad=new TPad("newpad","a transparent pad",0,0,1,1);
root[] newpad.SetFillStyle(4000);
root[] newpad.Draw();
root[] newpad.cd();
root[] // create some primitives, etc
```

**Setting the Log Scale is a Pad Attribute**

Setting the scale to logarithmic or linear is an attribute of the pad, not the axis or the histogram. The scale is an attribute of the pad because you may want to draw the same histogram in linear scale in one pad and in log scale in another pad. Frequently, we see several histograms on top of each other in the same pad. It would be very inconvenient to set the scale attribute for each histogram in a pad. Furthermore, if the logic were in the histogram class (or each object), one would have to test for the scale setting in each the Paint methods of all objects.

If you have a pad with a histogram, a right-click on the pad, outside of the histograms frame will convince you. The `SetLogx()`, `SetLogy()` and `SetLogz()` methods are there. As you see, `TPad` defines log scale for the two directions $x$ and $y$ plus $z$ if you want to draw a 3D representation of some function or histogram.

The way to set log scale in the $x$ direction for the active pad is:

```c
root[] gPad->SetLogx(1)
```

To reset log in the $z$ direction:

```c
root[] gPad->SetLogz(0)
```

If you have a divided pad, you need to set the scale on each of the sub-pads. Setting it on the containing pad does not automatically propagate to the sub-pads. Here is an example of how to set the log scale for the $x$-axis on a canvas with four sub-pads:

```c
root[] TCanvas MyCanvas("MyCanvas", "My Canvas")
root[] MyCanvas->Divide(2,2)
root[] MyCanvas->cd(1)
root[] gPad->SetLogx()
root[] MyCanvas->cd(2)
root[] gPad->SetLogx()
root[] MyCanvas->cd(3)
root[] gPad->SetLogx()
```

**Locking The Pad**

You can make the `TPad` non-editable. Then no new objects can be added, and the existing objects and the pad can not be changed with the mouse or programmatically.

```c
TPad::setEditable(kFALSE)
```

By default the `TPad` is editable.

---

**Graphical Objects**

In this paragraph, we describe the various simple 2D graphical objects defined in ROOT. Usually, one defines these objects with their constructor and draws them with their `Draw()` method. Therefore, the examples will be very brief. Most graphical objects have line and fill attributes (color, width) that will be described in "Graphical objects attributes".

If the user wants more information, the class names are given and he may refer to the online developer documentation. This is especially true for functions and methods that set and get internal values of the objects described here.

By default 2D graphical objects are created in User Coordinates with 0,0 in the lower left corner.

**Lines, Arrows, and Geometrical Objects**

**Line: Class TLine**

The simplest graphical object is a line. It is implemented in the `TLine` class.

The constructor is:

```c
TLine(Double_t x1, Double_t y1, Double_t x2, Double_t y2)
```

The arguments $x_1$, $y_1$, $x_2$, $y_2$ are the coordinates of the first and second point.

This constructor may be used as in:

```c
root[] 1 = new TLine(0.2,0.2,0.8,0.3)
root[] 1->Draw()
```

**Arrows: Class TArrow**

Different arrow formats as show in the picture below are available.

Once an arrow is drawn on the screen, one can:

- click on one of the edges and move this edge.
- click on any other arrow part to move the entire arrow.

The constructor is:

```c
TArrow(Double_t x1, Double_t y1, Double_t x2, Double_t y2, Float_t arrowsize, Option_t *option)
```
It defines an arrow between points $x_1, y_1$ and $x_2, y_2$. The arrow size is in percentage of the pad height. The options are the following:

- option = "$>"  
- option = "$<"  
- option = "$|>"  
- option = "$<|"  
- option = "$<\$>"  
- option = "$<|>"  

If FillColor == 0, draw open triangle else draw full triangle with fill color. If $ar$ is an arrow object, fill color is set with:

```cpp
ar.SetFillColor(icolor);
```

Where $icolor$ is the color defined in "Color and color palettes".

The opening angle between the two sides of the arrow is 60 degrees. It can be changed with $ar->SetAngle(angle)$, where angle is expressed in degrees.

**Poly-line: Class TPolyLine**

A poly-line is a set of joint segments. It is defined by a set of $N$ points in a 2D space. Its constructor is:

```cpp
TPolyLine(Int_t n, Double_t* x, Double_t* y, Option_t* option)
```

Where $n$ is the number of points, and $x$ and $y$ are arrays of $n$ elements with the coordinates of the points.

$TPolyLine$ can be used by itself, but is also a base class for other objects, such as curly arcs.

**Circles, Ellipses: Class TEllipse**

Ellipse is a general ellipse that can be truncated and rotated. An ellipse is defined by its center $(x_1, y_1)$ and two radii $r_1$ and $r_2$. A minimum and maximum angle may be specified (phimin, phimax). The picture below illustrates different types of ellipses:

The attributes of the outline line and of the fill area are described in "Graphical objects attributes".

An ellipse may be created with a statement like:

```cpp
root [] e = new TEllipse(0.2,0.2,0.8,0.3)
root [] e->Draw()
```

**Rectangles: Classes TBox and TWbox**

A rectangle is defined by the class $TBox$ since it is a base class for many different higher-level graphical primitives.

A box is defined by its bottom left coordinates $x_1$, $y_1$ and its top right coordinates $x_2$, $y_2$.

The constructor being:

```cpp
TBox(Double_t x1, Double_t y1, Double_t x2, Double_t y2)
```

It may be used as in:

```cpp
root [] b = new TBox(0.2,0.2,0.8,0.3)
root [] b->Draw()
```

A $TWbox$ is a rectangle ($TBox$) with a border size and a border mode.

The attributes of the outline line and of the fill area are described in "Graphical Objects Attributes".
One Point, or Marker: Class TMarker

A marker is a point with a fancy shape! The possible markers are the following:

One marker is build via the constructor:

```
TMarker(Double_t x, Double_t y, Int_t marker)
```

The parameters \( x \) and \( y \) are the coordinates of the marker and \( \text{marker} \) is the type, shown above.

Suppose \( ma \) is a valid marker. One can set the size of the marker with

```
ma->SetMarkerSize(size)
```

where \( \text{size} \) is the desired size. The available sizes are:

Sizes smaller than 1 may be specified.

Set of Points: Class TPolyMarker

A TPolyMarker is defined by an array on \( N \) points in a 2-D space. At each point \( x[i], y[i] \) a marker is drawn. The list of marker types is shown in the previous paragraph.

The marker attributes are managed by the class TAttMarker and are described in “Graphical objects attributes”

The constructor for a TPolyMarker is:

```
TPolyMarker(Int_t n, Double_t *x, Double_t *y, Option_t *option)
```

Where \( x \) and \( y \) are arrays of coordinates for the \( n \) points that form the poly-marker.

Curly and Wavy Lines for Feynman Diagrams

This is a peculiarity of particle physics, but we do need sometimes to draw Feynman diagrams. Our friends working in banking can skip this part.

A set of classes implements curly or wavy poly-lines typically used to draw Feynman diagrams. Amplitudes and wavelengths may be specified in the constructors, via commands or interactively from context menus. These classes are TCurlyLine and TCurlyArc.

These classes make use of TPolyLine by inheritance: ExecuteEvent methods are highly inspired from the methods used in TPolyLine and TArc.

The picture below has been generated by the tutorial `feynman.C`

### CurlyLine

The constructors are:

```
TCurlyLine(Double_t x1, Double_t y1, Double_t x2, Double_t y2, Double_t wavelength, Double_t amplitude)
```

With the starting point \((x1, y1)\), end point \((x2, y2)\), the wavelength and amplitude are given in percent of the pad height

For TCurlyArc, the constructor is:

```
TCurlyArc(Double_t x1, Double_t y1, Double_t rad, Double_t phimin, Double_t phimax, Double_t wavelength, Double_t amplitude)
```

The center is \((x1, y1)\) and the radius \(\text{rad}\). The wavelength and amplitude are given in percent of the line length, \(\text{phimin}\) and \(\text{phimax}\), which are the starting and ending angle of the arc, are given in degrees.

Refer to `$ROOTSYS/tutorials/feynman.C` for the script that built the picture above.

Text and Latex Mathematical Expressions

Text displayed in a pad may be embedded into boxes, called paves (such as PaveLabels), or titles of graphs or many other objects but it can live a life of its own. All text displayed in ROOT graphics is an object of class TText. For a physicist, it will be most of the time a TLatex expression (which derives from TText).

TLatex has been conceived to draw mathematical formulae or equations. Its syntax is very similar to the LaTeX one in mathematical mode.

Subscripts and Superscripts

Subscripts and superscripts are made with the \( _ \) and \( ^ \) commands. These commands can be combined to make complicated subscript and superscript expressions. You may choose how to display subscripts and superscripts using the 2 functions SetIndicesSize(Double_t) and SetLimitIndicesSize(Int_t).
Examples of what can be obtained using subscripts and superscripts:

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x^2y$</td>
<td>$x^2y$</td>
</tr>
<tr>
<td>$x_{2y}$</td>
<td>$x_{2y}$</td>
</tr>
<tr>
<td>$x^{y^{2}}$</td>
<td>$x^{y^{2}}$</td>
</tr>
<tr>
<td>$x^{y}_1$</td>
<td>$x^{y}_1$</td>
</tr>
</tbody>
</table>

Fractions

Fractions denoted by the `/` symbol are made in the obvious way. The \texttt{\textbackslash{frac}} command is used for large fractions in displayed formulas; it has two arguments: the numerator and the denominator. For example, this equation is obtained by following expression.

$$x = \frac{y+z/2}{y^2+1}$$

Roots

The \texttt{\textbackslash{sqrt}} command produces the square \textbf{R}OOT of its argument; it has an optional first argument for other roots.

Example: \texttt{\textbackslash{sqrt}[10]{10}} \texttt{\textbackslash{sqrt}[3]{10}}

Delimiters

You can produce three kinds of proportional delimiters.

\begin{align*}
&\{\ldots\} & \\
&\{\ldots\} & \\
&\{\ldots\} & \\
&\{\ldots\} & \\
&\{\ldots\}&
\end{align*}

Greek Letters

The command to produce a lowercase Greek letter is obtained by adding a \texttt{\textbackslash{a}} to the name of the letter. For an uppercase Greek letter, just capitalize the first letter of the command name.

$\texttt{\alpha \beta \gamma \Delta \epsilon \zeta \eta \theta \iota \kappa \lambda \mu \nu \xi \omicron \pi \rho \sigma \tau \upsilon \phi \varphi \chi \psi \omega \Gamma \Delta \Theta \Lambda \Xi \Pi \Sigma \Upsilon \Phi \Psi \Omega}$

Changing Style in Math Mode

You can change the font and the text color at any moment using:

\begin{align*}
&\texttt{\textbackslash{font}[\textbackslash{font-number}]{\ldots}} \quad \texttt{\textbackslash{color}[\textbackslash{color-number}]{\ldots}}
\end{align*}

Mathematical Symbols

\texttt{T\LaTeX} can make mathematical and other symbols. A few of them, such as "+" and ">", are produced by typing the corresponding keyboard character. Others are obtained with the commands in the following table.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\leq$</td>
<td>\texttt{\textbackslash{leq}}</td>
</tr>
<tr>
<td>$\geq$</td>
<td>\texttt{\textbackslash{geq}}</td>
</tr>
<tr>
<td>$\nabla$</td>
<td>\texttt{\textbackslash{nabla}}</td>
</tr>
<tr>
<td>$\approx$</td>
<td>\texttt{\textbackslash{approx}}</td>
</tr>
</tbody>
</table>

Accents, Arrows and Bars

Symbols in a formula are sometimes placed one above another. \texttt{T\LaTeX} provides special commands for doing this.

\begin{align*}
&\texttt{\textbackslash{hat}{a}} = \hat{a} \\
&\texttt{\textbackslash{check}{a}} = \check{a} \\
&\texttt{\textbackslash{acute}{a}} = \acute{a} \\
&\texttt{\textbackslash{grave}{a}} = \grave{a} \\
&\texttt{\textbackslash{dot}{a}} = \dot{a} \\
&\texttt{\textbackslash{ddot}{a}} = \ddot{a} \\
&\texttt{\textbackslash{tilde}{a}} = \tilde{a} \\
&\texttt{\textbackslash{slash}{a}} = \slash{a}
\end{align*}
Example 1
The following script ($ROOTSYS/tutorials/latex.C$):

```c
{ gROOT->Reset();
  TCanvas c1("c1","Latex",600,700);
  TLatex l;
  l.SetTextAlign(12);
  l.SetTextSize(0.04);
  l.DrawLatex(0.1,0.8,\"1) C(x) = d \sqrt{\frac{2}{\lambda D}} \int^{x}_{0} \cos(\frac{\pi}{2}t^{2})dt\");
  l.DrawLatex(0.1,0.6,\"2) C(x) = d \sqrt{\frac{2}{\lambda D}} \int^{x} \cos(\frac{\pi}{2}t^{2})dt\");
  l.DrawLatex(0.1,0.4,\"3) \ R = |A|^2 = \frac{1}{2}(\frac{1}{2}+C(V)^2+\frac{1}{2}+S(V)^2)\");
  l.DrawLatex(0.1,0.2,\"4) \ F(t) = \sum_{i=-\infty}^{\infty} A(i) \cos(\frac{\pi}{2}t+i)\");
}
```

The script makes this picture:

---

Example 2
The following script ($ROOTSYS/tutorials/latex2.C$):

```c
{ gROOT->Reset();
  TCanvas c1("c1","Latex",600,700);
  TLatex l;
  l.SetTextAlign(23);
  l.SetTextSize(0.1);
  l.DrawLatex(0.5,0.95,\"e^+e^- \rightarrow Z^0 \rightarrow I\bar{I}, q\bar{q}\");
  l.DrawLatex(0.5,0.75,\"|\vec{a}\cdot\vec{b}|=\Sigma a^{i}_{jk}+b^{bj}_{i}\");
  l.DrawLatex(0.5,0.5,\"i(\partial_{\mu}\bar{\psi}\gamma^{\mu}\psi + m\bar{\psi} = 0 \iff (\Box+m^2)\psi = 0\");
  l.DrawLatex(0.5,0.3,\"L_{em}=eJ^{\mu}_{em}A_{\mu}, \quad J^{\mu}_{em}=\bar{I}\gamma_{\mu}I, \quad M^{j}_{i}=(\Sigma A_{\alpha})^{j}_{i}\");
}
```

The result is the following picture:
Example 3
The following script ($ROOTSYS/tutorials/latex3.C$):

```c
{gROOT->Reset();
 TCanvas c1("c1");
 TPaveText pt(.1,.5,.9,.9);
 pt.AddText("{2s}/\pi \alpha^2 \frac{d\sigma}{d\cos{\theta}} (e^+e^- \rightarrow f\bar{f}) = ");
 pt.AddText("|1/(1-\Delta\alpha)|^2 (1+\cos^2{\theta} + 4 \Re \left\{ \frac{2}{1-\Delta\alpha} \chi(s) \right\} [g_{\nu}^e g_{\nu}^f (1+\cos^2{\theta}) + 2 g_a^e g_a^f \cos{\theta}] \right\} = ");
 pt.SetLabel("Born equation");
 pt.Draw();
}
```

The result is the following picture:

The result is the following picture:

![Born equation](image)

**Text in Labels and TPaves**

Text displayed in a pad may be embedded into boxes, called paves, or may be drawn alone. In any case, it is recommended to use a LaTeX expression, which is covered in the previous paragraph. Using TLatex is valid whether the text is embedded or not. In fact, you will use LaTeX expressions without knowing it since it is the standard for all the embedded text.

A pave is just a box with a border size and a shadow option. The options common to all types of paves and used when building those objects, are the following:

- **Option = “T”** Top frame
- **Option = “B”** Bottom frame
- **Option = “R”** Right frame
- **Option = “L”** Left frame
- **Option = “NDC”** x1,y1,x2,y2 are given in NDC
- **Option = “ARC”** corners are rounded

We will see the practical use of these options in the description of the more functional objects like TPaveLabels.

There are several categories of paves containing text:

**TPaveLabels**

TPaveLabels are panels containing one line of text. They are used for labeling. The constructor is:

```c
TPaveLabel(Double_t x1, Double_t y1,Double_t x2, Double_t y2, const char *label, Option_t *option)
```

Where (x1, y1) are the coordinates of the bottom left corner, (x2, y2) the coordinates of the upper right corner. "label" is the text to be displayed and "option" is the drawing option, described above. By default, the border size is 5 and the option is "br".

If one wants to set the border size to some other value, one may use the SetBorderSize() method. For example, suppose we have a histogram, which limits are (-100, 100) in the x direction and (0,1000) in the y direction.

The following lines will draw a label in the center of the histogram, with no border. If one wants the label position to be independent of the histogram coordinates, or user coordinates, one can use the option "NDC". See the paragraph about coordinate systems for more information.

```c
root[] pl = new TPaveLabel(-50, 0, 50,200,"Some text")
root[] pl->SetBorderSize(0)
root[] pl->Draw()
```
Here are examples of what may be obtained:

![Examples of TPaveText and TPaveLabel](image)

**TPaveText**

A TPaveText can contain only one line of text. A TPaveText may contain several lines. This is the only difference. This picture illustrates and explains some of the points of TPaveText. Once a TPaveText is drawn, a line can be added or removed by bringing up the context menu with the mouse.

```cpp
TPavesText(Double_t x1, Double_t y1, Double_t x2, Double_t y2, Int_t npaves, Option_t* option)
```

By default, the number of stacked panels is 5 and option = "br"

**Sliders**

Sliders may be used for showing the evolution of a process or setting the limits of an object's value interactively. A TSlider object contains a slider box that can be moved or resized.

Slider drawing options include the possibility to change the slider starting and ending positions or only one of them.

The current slider position can be retrieved via the functions `TSlider::GetMinimum()` and `TSlider::GetMaximum()`. These two functions return numbers in the range [0,1).

One may set a C expression to be executed when the mouse button 1 is released. This is done with the `TSlider::SetMethod()` function. It is also possible to reference an object. If no method or C expression has been set, and an object is referenced (setObject has been called), while the slider is being moved/resized, the object `ExecuteEvent` function is called.
Let's see an example using `SetMethod`. The script is called `xyslider.C`.
You can find this script in `$ROOTSYS/tutorials`.

```c
// Example of script featuring two sliders
TFile *f = new TFile("hsimple.root");
TH2F *hxpy = (TH2F*)f->Get("hxpy");
TCanvas *c1 = new TCanvas("c1");
TPad *pad = new TPad("pad", "lego pad", 0.1, 0.1, 0.98, 0.98);
pad->SetFillColor(33);
pad->Draw();
pad->cd();
gStyle->SetFrameFillColor(42);
hxpy->SetFillColor(46);
hxpy->Draw("lego1");
c1->cd();

// Create two sliders in main canvas. When button1
// of the mouse will be released, action.C will be called
TSlider *xslider = new TSlider("xslider", "x", .1, .02, .98, .08);
xslider->SetMethod(".x action.C");
TSlider *yslider = new TSlider("yslider", "y", .02, .1, .06, .98);
yslider->SetMethod(".y action.C");
}
```

The script that is executed when button 1 is released is the following (script `action.C`):

```c
Int_t nx = hxpy->GetXaxis()->GetNbins();
Int_t ny = hxpy->GetYaxis()->GetNbins();
Int_t binnxmin = nx*xslider->GetMinimum();
Int_t binnxmax = nx*xslider->GetMaximum();
hxpy->GetXaxis()->SetRange(binnxmin, binnxmax);
Int_t binnymmin = ny*yslider->GetMinimum();
Int_t binnymmax = ny*yslider->GetMaximum();
hxpy->GetYaxis()->SetRange(binnymax, binnmax);

pad->cd();
hxpy->Draw("lego1");
c1->Update();
```

The canvas and the sliders created in the above script are shown in the picture below.

The second example uses `SetObject` (script `xyslider.C`). Same example as above but using the `SetMethod`:

```c
Myclass *obj = new Myclass();
TSlider xslider->SetObject(obj);
yslider->SetObject(obj);
```

When one of the sliders will be changed, `Myclass::ExecuteEvent()` will be called with `px=0` and `py = 0`.

**Axis**

The axis objects are automatically built by various high level objects such as histograms or graphs. Once build, one may access them and change their characteristics. It is also possible, for some particular purposes to build axis on their own. This may be useful for example in the case one wants to draw two axis for the same plot, one on the left and one on the right.

For historical reasons, there are two classes representing axis:

- **TAxis** is the axis object, which will be returned when calling the `TH1::GetAxis()` method.

- **TAxis** is the axis object, which will be returned when calling the `TH1::GetAxis()` method.

Of course, you may do the same for Y and Z-axis.

The graphical representation of an axis is done with the `TGaxis` class. Instances of this class are generated by the histogram classes and `TGraph`. This is internal and the user should not have to see it.

**Axis Title**

The axis title is set, as with all named objects, by

```c
axis->SetTitle("Whatever title you want");
```

When the axis is embedded into a histogram or a graph, one has to first extract the axis object:

```c
h->GetXaxis()->SetTitle("Whatever title you want");
```
Axis Options and Characteristics

The axis options are most simply set with the styles. The available style options controlling specific axis options are the following:

- `TAxis *axis = histo->GetXaxis();`
- `axis->SetAxisColor(Color_t color = 1);`
- `axis->SetLabelColor(Color_t color = 1);`
- `axis->SetLabelFont(Style_t font = 62);`
- `axis->SetLabelOffset(Float_t offset = 0.005);`
- `axis->SetLabelSize(Float_t size = 0.04);`
- `axis->SetNdivisions(Int_t n = 510, Bool_t optim = kTRUE);`
- `axis->SetTitleOffset(Float_t offset = 1);`
- `axis->SetTitleSize(Float_t size = 0.02);`

The getters corresponding to the described setters are also available. Furthermore, the general options, not specific to axis, as for instance `SetTitleTextColor()` are valid and do have an effect on axis characteristics.

Setting the Number of Divisions

To set the number of divisions for an axis use

- `TAxis::SetNdivisions(ndiv, optim)` where `ndiv` and `optim` are as follows:
  - `ndiv = N1 + 100*N2 + 10000*N3`
  - `N1` = number of first divisions.
  - `N2` = number of secondary divisions.
  - `N3` = number of tertiary divisions.
  - `optim = kTRUE` (default), the number of divisions will be optimized around the specified value.
  - `optim = kFALSE`, or `n < 0`, the axis will be forced to use exactly `n` divisions.

For example:

- `ndiv = 0` : no tick marks.
- `ndiv = 2` : 2 divisions, one tick mark in the middle of the axis.
- `ndiv = 510` : 10 primary divisions, 5 secondary divisions
- `ndiv = -10` : exactly 10 primary divisions

Zooming the Axis

You can use `TAxis::SetRange` or `TAxis::SetRangeUser` to zoom the axis.

`TAxis::SetRange(Int_t binfirst, Int_t binlast)`

The `SetRange` method parameters are bin numbers. They are not axis. For example if a histogram plots the values from 0 to 500 and has 100 bins, `SetRange(0,10)` will cover the values 0 to 50.

Drawing Axis independently of Graphs or Histograms

An axis may be drawn independently of a histogram or a graph. This may be useful to draw for example a supplementary axis for a graph. In this case, one has to use the `TGaxis` class, the graphical representation of an axis.

The constructor for this kind of objects:

- `TGaxis(Double_t xmin, Double_t ymin, Double_t xmax, Double_t ymax, Double_t wmin, Double_t wmax, Int_t ndiv = 510, Option_t* chopt, Double_t gridlength = 0)`

The arguments `xmin`, `ymin` are the coordinates of the axis' start in the user coordinates system, and `xmax`, `ymax` are the end coordinates. The arguments `wmin` and `wmax` are the minimum (at the start) and maximum (at the end) values to be represented on the axis.

`ndiv` is the number of divisions (see above).

The options, given by the "chopt" string are the following:

- `chopt = 'G'`: logarithmic scale, default is linear.
- `chopt = 'B'`: Blank axis. Useful to superpose the axis.

Instead of the `wmin`, `wmax` arguments of the normal constructor, i.e. the limits of the axis, the name of a TF1 function can be specified. This function will be used to map the user coordinates to the axis values and ticks.

The constructor is the following:

- `TGaxis(Double_t xmin, Double_t ymin, Double_t xmax, Double_t ymax, const char* funcname, Int_t ndiv = 510, Option_t* chopt, Double_t gridlength = 0)`

In such a way, it is possible to obtain exponential evolution of the tick marks position, or even decreasing. In fact, anything you like.

Orientation of tick marks on axis.

Tick marks are normally drawn on the positive side of the axis, however, if `xmin = xmax`, then negative.

- `chopt = '+'`: tick marks are drawn on Positive side. (Default)
- `chopt = '-'`: tick marks are drawn on the negative side.
- `chopt = '+-'`: tick marks are drawn on both sides of the axis.
- `chopt = 'U'`: unlabeled axis, default is labeled.

The parameters for `SetRangeUser` are user coordinates. If the start or end is in the middle of a bin the resulting range is approximation. It finds the low edge bin for the start and the high edge bin for the high.
Label Position
Labels are normally drawn on side opposite to tick marks. However, \texttt{chopt = '='} on Equal side.

Label Orientation
Labels are normally drawn parallel to the axis. However, if $\text{xmin} = \text{xmax}$, then they are drawn orthogonal, and if $\text{ymin} = \text{ymax}$ they are drawn parallel.

Labels for Exponents
By default, an exponent of the form $10^N$ is used when the label values are either all very small or very large. One can disable the exponent by calling:
\begin{verbatim}
TAxis::SetNoExponent(kTRUE)
\end{verbatim}

Note that this option is implicitly selected if the number of digits to draw a label is less than the \texttt{fgMaxDigits} global member.

If you have set the property \texttt{SetNoExponent} in \texttt{TAxis} (via \texttt{TAxis::SetNoExponent(...)}, \texttt{TGaxis} will inherit this property. \texttt{TGaxis} is the class responsible for drawing the axis.

\texttt{SetNoExponent} is also available from the axis context menu.

Tick Mark Label Position
Labels are centered on tick marks. However, if $\text{xmin} = \text{xmax}$, then they are right adjusted.

- \texttt{chopt = 'R'}: labels are Right adjusted on tick mark (default is centered)
- \texttt{chopt = 'L'}: labels are left adjusted on tick mark.
- \texttt{chopt = 'C'}: labels are centered on tick mark.
- \texttt{chopt = 'M'}: In the Middle of the divisions.

Label Formatting
Blank characters are stripped, and then the label is correctly aligned. The dot, if last character of the string, is also stripped. In the following, we have some parameters, like tick marks length and characters height (in percentage of the length of the axis, in user coordinates).

The default values are as follows:

- Primary tick marks: 3.0 %
- Secondary tick marks: 1.5 %
- Third order tick marks: 0.75 %
- Characters height for labels: 4%
- Labels offset: 1.0 %

Stripping Decimals
Use the \texttt{TStyle::SetStripDecimals} to strip decimals when drawing axis labels. By default, the option is set to true, and \texttt{TGaxis::PaintAxis} removes trailing 0s after a dot in the axis labels, e.g.: \{0.0, 0.5, 1.0, 1.5, 2.0, 2.5,...\}

\begin{verbatim}
TStyle::SetStripDecimals(Bool_t strip=kTRUE)
\end{verbatim}

If this function is called with \texttt{strip=kFALSE}, \texttt{TGaxis::PaintAxis()} will draw labels with the same number of digits after the dot, e.g.: \{0.0, 0.5, 1.0, 1.5, 2.0, 2.5,...\}

Optional Grid
\texttt{chopt = 'W'}: cross-Wire

Axis Binning Optimization
By default, the axis binning is optimized.

- \texttt{chopt = 'N'}: No binning optimization
- \texttt{chopt = 'I'}: Integer labeling
Time Format

Axis labels may be considered as times, plotted in a defined time format. The format is set with `SetTimeFormat()`.

`chopt = 't'`: Plot times with a defined format instead of values

The format string for date and time use the same options as the one used in the standard `strftime` C function.

For the date:

- %a : abbreviated weekday name
- %b : abbreviated month name
- %d : day of the month (01-31)
- %m : month (01-12)
- %y : year without century

For the time:

- %H : hour (24-hour clock)
- %I : hour (12-hour clock)
- %p : local equivalent of AM or PM
- %M : minute (00-59)
- %S : seconds (00-61)
- %% : 

The start time of the axis will be `wmin + time offset`. This time offset is the same for all axes, since it is gathered from the active style. One may set the time offset:

```cpp
gStyle->SetTimeOffset(time)
```

Where "time" is the offset time expressed in UTC (Universal Coordinated Time) and is the number of seconds since a standard date (1970), adjusted for some earth's rotation drifting. Your computer time is using UTC as a reference.

**Axis Example 1:**

To illustrate all what was said before, we can show two scripts. This example creates this picture:
Axis Example 2:
The second example shows the use of the second form of the constructor, with axis ticks position determined by a function TF1:

```c
void gaxis3a()
{
    gStyle->SetOptStat(0);
    TH2F *h2 = new TH2F("h","Axes",2,0,10,2,-2,2);
    h2->Draw();
    TF1 *f1=new TF1("f1","-x",-10,10);
    TGaxis *A1 = new TGaxis(0,2,10,2,"f1",510,"-");
    A1->SetTitle("axis with decreasing values");
    A1->Draw();
    TF1 *f2=new TF1("f2","exp(x)",0,2);
    TGaxis *A2 = new TGaxis(1,1,9,1,"f2");
    A2->SetTitle("exponential axis");
    A2->SetLabelSize(0.03);
    A2->SetTitleSize(0.03);
    A2->SetTitleOffset(1.2);
    A2->Draw();
    TF1 *f3=new TF1("f3","log10(x)",0,800);
    TGaxis *A3 = new TGaxis(2,-2,2,0,"f3",505);
    A3->SetTitle("logarithmic axis");
    A3->SetLabelSize(0.03);
    A3->SetTitleOffset(1.2);
    A3->Draw();
}
```

Axis Example with Time display:

```c
// strip chart example
void seism() {
    TStopwatch sw; sw.Start();
    //set time offset
    TDatime dtime;
    gStyle->SetTimeOffset(dtime.Convert());
    TCanvas *c1 = new TCanvas("c1","Time on axis",
10,10,1000,500);
    c1->SetFillColor(42);
    c1->SetFrameFillColor(33);
    c1->SetGrid();
    Float_t bintime = 1;
    //one bin = 1 second. change it to set the time scale
    TH1F *ht = new TH1F("ht","The ROOT seism",
10,0,10*bintime);
    Float_t signal = 1000;
    ht->SetMaximum(signal);
    ht->SetMinimum(-signal);
    ht->SetStats(0);
    ht->SetLineColor(2);
    ht->GetXaxis()->SetTimeDisplay(1);
    ht->GetYaxis()->SetNdivisions(520);
    ht->Draw();
    for (Int_t i=1;i<2300;i++) {
        //======= Build a signal : noisy damped sine ======
        Float_t noise  = gRandom->Gaus(0,120);
        if (i > 700) noise += signal*sin((i-700.)*6.28/30)*exp((700.-i)/300.);
        ht->SetBinContent(i,noise);
        c1->Modified();
        c1->Update();
        gSystem->ProcessEvents();
        //canvas can be edited during the loop
    }
    printf("Real Time = %8.3fs,
     Cpu Time = %8.3fs\n",sw.RealTime(),sw.CpuTime());
}
```
Graphics and the Graphical User Interface  

**Graphical Objects Attributes**

**Text Attributes**

When a class contains text or derives from a text class, it needs to be able to set text attributes like font type, size, and color. To do so, the class inherits from the TAttText class (a secondary inheritance), which defines text attributes. TLatex and TText inherit from TAttText.

**Setting Text Attributes Interactively**

When clicking on an object containing text, one of the last items in the context menu is `SetTextAttributes`. Selecting it makes the following window appear:

![Setting Text Attributes Window](image)

This canvas allows you to set:

- The text alignment
- Font
- Color
- Size

**Setting Text Alignment**

Text alignment may also be set by a method call. What is said here applies to all objects deriving from TAttText, and there are many. We will take an example that may be transposed to other types. Suppose "la" is a TLatex object. The alignment is set with:

```
root[] la->SetTextAlign(align)
```

The parameter `align` is a short describing the alignment:

```
align = 10*HorizontalAlign + VerticalAlign
```

For Horizontal alignment the following convention applies:

- 1 = left
- 2 = centered
- 3 = right

For Vertical alignment the following convention applies:

- 1 = bottom
- 2 = centered
- 3 = top

For example

Align: 11 = left adjusted and bottom adjusted  
Align: 32 = right adjusted and vertically centered

**Setting Text Angle**

Use `TAttText::SetTextAngle` to set the text angle. The angle is the degrees of the horizontal.

```
root[] la->SetTextAngle(angle)
```

**Setting Text Color**

Use `TAttText::SetTextColor` to set the text color. The color is the color index. The colors are described in "Color and color palettes".

```
root[] la->SetTextColor(color)
```

**Setting Text Font**

Use `TAttText::SetTextFont` to set the font. The parameter font is the font code, combining the font and precision:

```
font = 10 * fontID + precision
```

The table below lists the available fonts. The font IDs must be between 1 and 14.

The precision can be:

- Precision = 0 fast hardware fonts (steps in the size)
• Precision = 1 scalable and rotate-able hardware fonts (see below)
• Precision = 2 scalable and rotate-able hardware fonts

When precision 0 is used, only the original non-scaled system fonts are used. The fonts have a minimum (4) and maximum (37) size in pixels. These fonts are fast and are of good quality. Their size varies with large steps and they cannot be rotated.

Precision 1 and 2 fonts have a different behavior depending if True Type Fonts (TTF) are used or not. If TTF are used, you always get very good quality scalable and rotate-able fonts. However, TTF are slow.

Precision 1 and 2 fonts have a different behavior for PostScript in case of \texttt{TLatex} objects:

• With precision 1, the PostScript text uses the old convention (see \texttt{TPostScript}) for some special characters to draw sub and superscripts or Greek text.
• With precision 2, the "PostScript" special characters are drawn as such. To draw sub and superscripts it is highly recommended to use \texttt{TLatex} objects instead.

For example: \texttt{font = 62} is the font with ID 6 and precision 2

The available fonts are:

<table>
<thead>
<tr>
<th>Font ID</th>
<th>X11</th>
<th>True Type name</th>
<th>is italic</th>
<th>&quot;boldness&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>times-medium-i-normal</td>
<td>&quot;Times New Roman&quot;</td>
<td>Yes</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>times-bold-r-normal</td>
<td>&quot;Times New Roman&quot;</td>
<td>No</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>times-bold-i-normal</td>
<td>&quot;Times New Roman&quot;</td>
<td>Yes</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>helvetica-medium-r-normal</td>
<td>&quot;Arial&quot;</td>
<td>No</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>helvetica-medium-o-normal</td>
<td>&quot;Arial&quot;</td>
<td>Yes</td>
<td>4</td>
</tr>
<tr>
<td>6</td>
<td>helvetica-bold-r-normal</td>
<td>&quot;Arial&quot;</td>
<td>No</td>
<td>7</td>
</tr>
<tr>
<td>7</td>
<td>helvetica-bold-o-normal</td>
<td>&quot;Arial&quot;</td>
<td>Yes</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>courier-medium-r-normal</td>
<td>&quot;Courier New&quot;</td>
<td>No</td>
<td>4</td>
</tr>
<tr>
<td>9</td>
<td>courier-medium-o-normal</td>
<td>&quot;Courier New&quot;</td>
<td>Yes</td>
<td>4</td>
</tr>
<tr>
<td>10</td>
<td>courier-bold-r-normal</td>
<td>&quot;Courier New&quot;</td>
<td>No</td>
<td>7</td>
</tr>
<tr>
<td>11</td>
<td>courier-bold-o-normal</td>
<td>&quot;Courier New&quot;</td>
<td>Yes</td>
<td>7</td>
</tr>
<tr>
<td>12</td>
<td>symbol-medium-r-normal</td>
<td>&quot;Symbol&quot;</td>
<td>No</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>times-medium-r-normal</td>
<td>&quot;Times New Roman&quot;</td>
<td>No</td>
<td>4</td>
</tr>
<tr>
<td>14</td>
<td>&quot;Wingdings&quot;</td>
<td></td>
<td>No</td>
<td>4</td>
</tr>
</tbody>
</table>

Here is an example of what the fonts look like:

This script makes the image of the different fonts:

\begin{verbatim}
{ textc = new TCanvas("textc","Example of text",1); for (int i=1;i<15;i++) {
  cid = new char[8];
  sprintf(cid,"ID %d :",i);
  cid[7] = 0;
  lid = new TLatex(0.1,1-(double)i/15,cid);
  lid->SetTextFont(62);
  lid->Draw();
  l = new TLatex(.2,1-(double)i/15,
"The quick brown fox is not here anymore");
  l->SetTextFont(i*10+2);
  l->Draw();
}
}\end{verbatim}

\textbf{How to use True Type Fonts}

You can activate the True Type Fonts by adding the following line in your \texttt{.rootrc} file.

\begin{verbatim}
Unix.*.Root.UseTTFonts: true
\end{verbatim}

You can check that you indeed use the TTF in your Root session. When the TTF is active, you get the following message at the start of a session:

"Free Type Engine v1.x used to render TrueType fonts."

You can also check with the command:

\begin{verbatim}
gEnv->Print()
\end{verbatim}
**Setting Text Size**

Use `TAttText::SetTextSize` to set the text size.

```cpp
root[] la->SetTextSize(size)
```

The size is the text size expressed in percentage of the current pad size. The text size in pixels will be:

- If current pad is horizontal, the size in pixels = `textsize * canvas_height`
- If current pad is vertical, the size in pixels = `textsize * canvas_width`

**Line Attributes**

All classes manipulating lines have to deal with line attributes. This is done by using secondary inheritance of the class `TAttLine`.

**Setting Line Attributes Interactively**

When clicking on an object being a line or having some line attributes, one of the last items in the context menu is `SetLineAttributes`. Selecting it makes the following window appear:

- **Setting Line Color**
  Line color may be set by a method call. What is said here applies to all objects deriving from `TAttLine`, and there are many (histograms, plots). We will take an example that may be transposed to other types. Suppose "li" is a `TLine` object. The line color is set with:
  ```cpp
  root[] li->SetLineColor(color)
  ```
  The argument `color` is a color number. The colors are described in "Color and Color Palettes".

- **Setting Line Style**
  Line style may be set by a method call. What is said here applies to all objects deriving from `TAttLine`, and there are many (histograms, plots). We will take an example that may be transposed to other types. Suppose "li" is a `TLine` object. The line style is set with:
  ```cpp
  root[] li->SetLineStyle(style)
  ```
  The argument `style` is one of:
  - 1=solid, 2=dash, 3=dash-dot, 4=dot-dot.

- **Setting Line Width**
  Line width may be set by a method call. What is said here applies to all objects deriving from `TAttLine`, and there are many (histograms, plots). We will take an example that may be transposed to other types. Suppose "li" is a `TLine` object. The line width is set with:
  ```cpp
  root[] li->SetLineWidth(width)
  ```
  The width is the width expressed in pixel units.

**Fill Attributes**

Almost all graphics classes have a fill area somewhere. These classes have to deal with fill attributes. This is done by using secondary inheritance of the class `TAttFill`.

**Setting Fill Attributes interactively**

When clicking on an object having a fill area, one of the last items in the context menu is `SetFillAttributes`. Selecting it makes the following window appear:

- **Setting Fill Color**
  The fill color is selected.

- **Setting Fill Style**
  The fill style is selected.

- **Setting Fill Width**
  The fill width is selected.
**g Fill Color**

Fill color may be set by a method call. What is said here applies to all objects deriving from `TAttFill`, and there are many (histograms, plots). We will take an example that may be transposed to other types. Suppose "h" is a `TH1F` (1 dim histogram) object. The histogram fill color is set with:

```c
root[] h->SetFillColor(color)
```

The color is a color number. The colors are described in "Color and color palettes".

**Setting Fill Style**

Fill style may be set by a method call. What is said here applies to all objects deriving from `TAttFill`, and there are many (histograms, plots). We will take an example that may be transposed to other types. Suppose "h" is a `TH1F` (1 dim histogram) object. The histogram fill style is set with:

```c
root[] h->SetFillStyle(style)
```

The convention for `style` is:

<table>
<thead>
<tr>
<th>Value</th>
<th>Style</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>hollow</td>
</tr>
<tr>
<td>1001</td>
<td>solid</td>
</tr>
<tr>
<td>2001</td>
<td>hatch style</td>
</tr>
<tr>
<td>3000 + pattern number</td>
<td>patterns</td>
</tr>
<tr>
<td>4000 to 4100</td>
<td>transparency, 4000 = fully transparent, 4100 = fully opaque.</td>
</tr>
</tbody>
</table>

The various patterns are represented here:

![Fill styles](image)

**Color and Color Palettes**

At initialization time, a table of basic colors is generated when the first Canvas constructor is called. This table is a linked list, which can be accessed from the `gROOT` object (see `TROOT::GetListOfColors()`).

Each color has an index and when a basic color is defined, two "companion" colors are defined:
- The dark version (color_index + 100)
- The bright version (color_index + 150)

The dark and bright colors are used to give 3-D effects when drawing various boxes (see `TWbox`, `TPave`, `TPaveText`, `TPaveLabel`, etc).

If you have a black and white copy of the manual, here are the basic colors and their indices:

<table>
<thead>
<tr>
<th>Color Index</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>black</td>
</tr>
<tr>
<td>2</td>
<td>red</td>
</tr>
<tr>
<td>3</td>
<td>bright green</td>
</tr>
<tr>
<td>4</td>
<td>bright blue</td>
</tr>
<tr>
<td>5</td>
<td>yellow</td>
</tr>
<tr>
<td>6</td>
<td>hot pink</td>
</tr>
<tr>
<td>7</td>
<td>aqua</td>
</tr>
<tr>
<td>8</td>
<td>green</td>
</tr>
<tr>
<td>9</td>
<td>blue</td>
</tr>
<tr>
<td>10 to 19</td>
<td>shades of gray</td>
</tr>
<tr>
<td>20 to 29</td>
<td>shades of brown</td>
</tr>
<tr>
<td>30 to 39</td>
<td>shades of blue</td>
</tr>
<tr>
<td>40 to 49</td>
<td>shades of red</td>
</tr>
</tbody>
</table>

The list of currently supported basic colors (here dark and bright colors are not shown) is shown in the picture below:

The color numbers specified in the basic palette, and the picture above, can be viewed by selecting the item "Colors" in the "View" menu of the canvas toolbar.

Other colors may be defined by the user. To do this, one has to build a new object of type `TColor`, which constructor is:

```c
TColor(Int_t color, Float_t r, Float_t g, Float_t b, const char* name)
```

One has to give the color number and the three Red, Green, Blue values, each being defined from 0 (min) to 1 (max). An optional name may be given. When built, this color is automatically added to the existing list of colors.

If the color number already exists, one has to extract it from the list and redefine the RGB values. This may be done for example with:

```c
root[] color = (TColor*)(gROOT->GetListOfColors()-
                      At(index_color))
root[] color->SetRGB(r,g,b)
```

Where `r`, `g`, and `b` go from 0 to 1 and `index_color` is the color number you wish to change.

**Color Palette (for Histograms)**

Defining one color at a time may be tedious. The color palette is used by the histogram classes (see Draw Options). For example, `TH1::Draw("col")` draws a 2-D histogram with cells represented by a box filled with a color `CI` function of the cell content. If the cell content is `x`, the color `CI` used will be...
The Graphical Editor

ROOT has a built-in graphics editor to draw and edit graphic primitives starting from an empty canvas or on top of a picture (e.g. histogram). The editor is started by selecting the "Editor" item in the canvas "Edit" menu. A menu appears into an independent window.

You can create the following graphical objects:

- An arc or circle: Click on the center of the arc, and then move the mouse. A rubber band circle is shown. Click again with the left button to freeze the arc.
- A line or an arrow: Click with the left button at the point where you want to start the arrow, then move the mouse 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 context menu, 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 Pave Label: Proceed like for a pad. Type the text to be put in the box. Then type a carriage return. The text will be redrawn to fill the box.
- A Pave Text or Paves Text: Proceed like for a pad. You can then click on the TPaveText object with the right mouse button and select the option AddText.

A Poly Line: Click with the left button for the first point, move the mouse, click again with the left button for a new point. Close the poly-line with a double click. To edit one vertex point, pick it with the left button and drag it to its new position.

A Curly Line: Proceed as for the arrowline. Once done, click with the third button to change the characters of the curvy line, like transform it to wave, change the wavelength, etc...

A Curly Arc: Proceed like for the arrowline. The first click is located at the position of the center, the second click at the position of the arc beginning. Once done, one obtains a curvy ellipse, for which one can click with the third button to change the characters, like transform it to wavy, change the wavelength, set the minimum and maximum angle to make an arc that is not closed, etc...

A Text /Latex String: Click with the left button where you want to draw the text, then type in the text terminated by carriage return. All TLatex expressions are valid. To move the text or formula, point on it keeping the left mouse button down and drag it to its new position.

For example, to set the current palette to the "pretty" one, one has to do:

```cpp
color->SetRGB(1-(i/((colNum)*1.0)),0.3,0.5);
```
mouse button pressed and drag the text to its new position. You can
grow/shrink the text if you position the mouse to the first top-third part of the
string, then move the mouse up or down to grow or shrink the text
respectively. If you position the mouse near the bottom-end of the text, you
can rotate it.

A Marker: Click with the left button where to place the marker. The marker
can be modified by gStyle->SetMarkerStyle().

A Graphical Cut: Click with the left button on each point of a polygon
delimiting the selected area. Close the cut by double clicking on the last
point. A TCutG object is created. It can be used as a selection for a
TTree::Draw. You can get a pointer to this object with TCutG cut =
(TCutG*) gPad->GetPrimitive("CUTG").

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
script 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).

Copy/Paste With DrawClone

You can make a copy of a canvas using TCanvas::DrawClonePad. This
method is unique to TCanvas. It clones the entire canvas to the active pad.
There is a more general method TObject::DrawClone, which all objects
descendents of TObject, specifically all graphic objects inherit. Below are
two examples, one to show the use of DrawClonePad and the other to show
the use of DrawClone.

Example 1: TCanvas::DrawClonePad

In this example we will copy a canvas to a new one with
DrawClonePad.

Run the script draw2dopt.C

    root [] .x tutorials/draw2dopt.C

This creates a canvas with 2D histograms. To make a copy of the canvas
follows these steps

Right-click on it to bring up the context menu.
Select DrawClonePad.

This copies the entire canvas and all its sub-pads to a new canvas. The
copied canvas is a deep clone, and all the objects on it are copies and
independent of the original objects. For instance, change the fill on one of the
original histograms, and the cloned histogram retains its attributes.

DrawClonePad will copy the canvas to the active pad; the target does not
have to be a canvas. It can also be a pad on a canvas.

Example 2: TObject::DrawClone

If you want to copy and paste a graphic object from one canvas or pad to
another canvas or pad, you can do so with DrawClone method inherited
from TObject. The TObject::DrawClone method is inherited by all
graphic objects.

In this example, we create a new canvas with one histogram from each of the
canvases from the script draw2dopt.C.

1. Start a new ROOT session and execute the script draw2dopt.C
2. Select a canvas displayed by the script, and create a new canvas from
   the File menu (c1).
3. Make sure that the target canvas (c1) is the active one by middle
   clicking on it. If you do this step right after step 2, c1 will be active.
4. Select the pad with the first histogram you want to copy and paste.
5. Right click on it to show the context menu, and select DrawClone.
6. Leave the option blank and hit OK.

Repeat these steps for one histogram on each of the canvases created by
the script, until you have one pad from each type.
If you wanted to put the same annotation on each of the sub pads in the new canvas, you could use **DrawClone** to do so. Here we added the date to each pad. The steps to this are:

1. Create the label in one of the pads with the graphics editor.
2. Middle-click on the target pad to make it the active pad.
3. Use **DrawClone** method of the label to draw it in each of the other panels.

The option in the **DrawClone** method argument is the Draw option for a histogram or graph. A call to **TH1::DrawClone** can clone the histogram with a different draw option.

### Copy/Paste Programmatically

To copy and paste the four pads from the command line or in a script you would execute the following statements:

```c++
root [] .x tutorials/draw2dopt.C
root [] TCanvas c1("c1","Copy Paste",200,200,800,600);
root [] surfaces->cd(1);  // get the first pad
root [] TPad * p1 = gPad;
root [] lego->cd(2);       // get the next pad
root [] cont->cd(3);      // get the next pad
root [] TPad * p3 = gPad;
root [] c2b->cd(4);       // get the next pad
root [] TPad * p4 = gPad;  // draw the four clones
root [] c1->cd();
root [] p1->DrawClone();
root [] p2->DrawClone();
root [] p3->DrawClone();
root [] p4->DrawClone();
```

Note that the pad is copied to the new canvas in the same location as in the old canvas. For example if you were to copy the third pad of `surf` to the top left corner of the target canvas you would have to reset the coordinates of the cloned pad.

### Legends

Legends for a graph are obtained with a **TLegend** object. This object points to markers/lines/boxes/histograms/graphs and represent their marker/line/fill attribute. Any object that has a marker or line or fill attribute may have an associated legend.

A **TLegend** is a panel with several entries (class **TLegendEntry**) and is created by the constructor

```c++
TLegend( Double_t x1, Double_t y1,Double_t x2, Double_t y2,
const char *header, Option_t *option)
```

The legend is defined with default coordinates, border size and option. `x1,y1,x2,y2` are the coordinates of the legend in the current pad (in NDC coordinates by default). The default text attributes for the legend are:

- Alignment = 12 left adjusted and vertically centered
- Angle = 0 (degrees)
- Color = 1 (black)
- Size = calculate when number of entries is known
- Font = helvetica-medium-r-normal scalable font = 42, and bold = 62 for title

The title is a regular entry and supports **TLatex**. The default is no title (`header = 0`). The options are the same as for **TPave**; by default, they are "brNDC".

Once the legend box is created, one has to add the text with the **AddEntry()** method:

```c++
TLegendEntry* TLegend::AddEntry(TObject *obj, const char
*label, Option_t *option)
```

The parameters are:

- `*obj`: is a pointer to an object having marker, line, or fill attributes (for example a histogram, or graph)
- `label`: is the label to be associated to the object
- `option`:
  - o "L" draw line associated with line attributes of obj if obj has them (inherits from **TAttLine**)
  - o "P" draw poly-marker associated with marker attributes of obj if obj has them (inherits from **TAttMarker**)
  - o "F" draw a box with fill associated with fill attributes of obj if obj has them (inherits **TAttFill**)

One may also use the other form of **AddEntry**:

```c++
TLegendEntry* TLegend::AddEntry(const char *name, const
char *label, Option_t *option)
```

Where `name` is the name of the object in the pad. Other parameters are as in the previous case.
Here's an example of a legend created with TLegend:

```c
leg = new TLegend(0.4,0.6,0.89,0.89);
leg->AddEntry(fun1,"One Theory","l");
leg->AddEntry(fun3,"Another Theory","f");
leg->AddEntry(gr,"The Data","p");
leg->Draw();
// oops we forgot the blue line... add it after
leg->AddEntry(fun2,
  "#sqrt{2#pi} P_{T} (#gamma) latex formula","f");
// and add a header (or "title") for the legend
leg->SetHeader("The Legend Title");
leg->Draw();
```

Where `fun1`, `fun2`, `fun3`, and `gr` are pre-existing functions and graphs. You can edit the TLegend by right clicking on it.

The legend part of this plot was created as follows:

```c
pad1->Print("xxx.ps")
```

This prints the picture in the pad pointed by `pad1`. The TPad::Print method has a second parameter called option. Its value can be:

- 0 which is the default and is the same as "ps"
- "ps" a Postscript file is produced
- "eps" an Encapsulated Postscript file is produced
- "gif" a GIF file is produced
- "cxx" a C++ macro file is produced

You do not need to specify the second parameter, you can indicate by the filename extension what format you want to save a canvas in (i.e. `canvas.ps`, `canvas.gif`, etc).

The size of the PostScript picture, by default, is computed to keep the aspect ratio of the picture on the screen, where the size along x is always 20 cm.

You can set the size of the PostScript picture before generating the picture with a command such as:

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

The first parameter in the TPostScript constructor is the name of the file.
The second parameter is the format option.

- 111 - ps portrait
- 112 - ps landscape
- 113 - eps

You can set the default paper size with:

```c
gStyle->SetPaperSize(xsize,ysize);
```

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

To add text to a postscript file, use the method TPostScript::Text(x,y,"string"). This method writes the string in quotes into a PostScript file at position x, y in world coordinates.

**Special Characters**

The following characters have a special action on the PostScript file:

- \`: Go to Greek
- `: Go to special
- \`: Go to Zapf Dingbats
- ? : Go to subscript
- `^: Go to superscript
- \!: go to normal level of script
These special characters are printed as such on the screen. To generate one of these characters on the PostScript file, you must escape it with the escape character "@".

The use of these special characters is illustrated in several scripts referenced by the TPostScript constructor.

**Multiple Pictures in a PostScript File: Case 1**

The following script is an example illustrating how to open a PostScript file and draw several pictures. The generation of a new PostScript page is automatic when TCanvas::Clear is called by object->Draw().

```cpp
{    
    TFile f("hsimple.root");
    TCanvas c1("c1","canvas",800,600);
    //select PostScript output type
    Int_t type = 111; //portrait ps
    // Int_t type = 112; //landscape ps
    // Int_t type = 113; //eps
    //create a PostScript file and set the paper size
    TPostScript ps("test.ps",type);
    ps.Range(16,24); //set x,y of printed page
    //draw 3 histograms from file hsimple.root on separate pages
    hpx->Draw();
    c1.Update(); //force drawing in a script
    hprof->Draw();
    c1.Update();
    hpx->Draw("lego1");
    c1.Update();
    ps.Close();
}
```

**Multiple Pictures a PostScript File: Case 2**

This example shows 2 pages. The canvas is divided.

```cpp
{    
    TFile *f1 = new TFile("hsimple.root");
    TCanvas *c1 = new TCanvas("c1");
    TPostScript *ps = new TPostScript("file.ps",112);
    c1->Divide(2,1);
    // picture 1
    ps->NewPage();
    c1->cd(1);
    hpx->Draw();
    c1->cd(1);
    hprof->Draw();
    c1->Update();
    // picture 2
    ps->NewPage();
    c1->cd(1);
    hpxy->Draw();
    c1->cd(2);
    ntuple->Draw("px");
    c1->Update();
    ps->Close();
    // invoke PostScript viewer
    gSystem->Exec("gs file.ps");
}
```

**Create or Modify a Style**

All objects that can be drawn in a pad inherit from one or more attribute classes like TAttLine, TAttFill, TAttText, TAttMarker. When the objects are created, their default attributes are taken from the current style. The current style is an object of the class TStyle and can be referenced via the global variable gStyle (in TStyle.h). See the class TStyle for a complete list of the attributes that can be set in one style. ROOT provides several styles called

- "default" The default style
- "Plain" The simple style (black and white)
- "Bold" Bolder lines
- "Video" Suitable for html output or screen viewing
The "default" style is created by:

```cpp
TStyle *default = new TStyle("Default","Default Style");
```

The "Plain" style can be used if you are working on a monochrome display or if you want to get a "conventional" PostScript output. As an example, these are the instructions in the ROOT constructor to create the "Plain" style.

```cpp
TStyle *plain = new TStyle("Plain","Plain Style (no colors/fill areas)");
plain->SetCanvasBorderMode(0);
plain->SetPadBorderMode(0);
plain->SetPadColor(0);
plain->SetCanvasColor(0);
plain->SetTitleColor(0);
```

You can set the current style with:

```cpp
gROOT->SetStyle(style_name);
```

You can get a pointer to an existing style with:

```cpp
TStyle *style = gROOT->GetStyle(style_name);
```

You can create additional styles with:

```cpp
TStyle *st1 = new TStyle("st1","my style");
st1->Set....
st1->cd(); // this becomes now the current style gStyle
```

In your rootlogon.C file, you can redefine the default parameters via statements like:

```cpp
gStyle->SetStatX(0.7);
gStyle->SetStatW(0.2);
gStyle->SetLabelOffset(1.2);
gStyle->SetLabelFont(72);
```

Note that when an object is created, its attributes are taken from the current style. For example, you may have created a histogram in a previous session and saved it in a file. Meanwhile, if you have changed the style, the histogram will be drawn with the old attributes. You can force the current style attributes to be set when you read an object from a file by calling `ForceStyle()` before reading the objects from the file.

```cpp
gROOT->ForceStyle();
```

When you call `gROOT->ForceStyle()` and read an object from a ROOT file, the objects method `UseCurrentStyle()` is called. The attributes saved with the object are replaced by the current style attributes. You can also call `myObject->UseCurrentStyle()` directly. For example if you have a canvas or pad with your histogram or any other object, you can force these objects to get the attributes of the current style with:

```cpp
canvas->UseCurrentStyle();
```

The description of the style functions should be clear from the name of the TStyle setters or getters. Some functions have an extended description, in particular:

- TStyle::SetLabelFont
- TStyle::SetLineStyleString: set the format of dashed lines.
- TStyle::SetOptStat
- TStyle::SetPalette to change the colors palette
- TStyle::SetTitleOffset

To set the "Default" style, you can call:

```cpp
gROOT->SetStyle("Default");
```

To get a pointer to the "Default" style, you can call:

```cpp
TStyle *default_style = gROOT->GetStyle("Default");
```
10 Folders And Tasks

Folders

A TFolder is a collection of objects visible and expandable in the ROOT object browser. Folders have a name and a title and are identified in the folder hierarchy by an "UNIX-like" naming convention. The base of all folders is //root. It is visible at the top of the left panel in the browser. The browsers shows several folders under //root.

Why Use Folders?

One reason to use folders is to reduce class dependencies and improve modularity. Each set of data has a producer class and one or many consumer classes. When using folders, the producer class places a pointer to the data into a folder, and the consumer class retrieves a reference to the folder. The consumer can access the objects in a folder by specifying the path name of the folder.

Here is an example of a folder's path name:

//root/Event/Hits/TCP

One does not have to specify the full path name. If the partial path name is unique, it will find it, otherwise it will return the first occurrence of the path.

How to Use Folders

Using folders means building a hierarchy of folders, posting the reference to the data in the folder by the producer, and creating a reference to the folder by the consumer.

Creating a Folder Hierarchy

To create a folder hierarchy you add the top folder of your hierarchy to //root. Then you add a folder to an existing folder with the TFolder::AddFolder method. This method takes two parameters: the name and title of the folder to be added. It returns a pointer of the newly created folder.
The code below creates the folder hierarchy shown in the browser.

```c++
// Add the top folder of my hierarchy to //root
TFolder *aliroot = gROOT->GetRootFolder()
  ->AddFolder("aliroot", "aliroot top level folders");

// Add the hierarchy to the list of browsables
gROOT->GetListOfBrowsables()->Add(aliroot, "aliroot");

// Create and add the constants folder
TFolder *constants = aliroot
  ->AddFolder("Constants", "Detector constants");

// Create and add the pdg folder to pdg
TFolder *pdg = constants
  ->AddFolder("DatabasePDG", "PDG database");

// Create and add the run folder
TFolder *run = aliroot
  ->AddFolder("Run", "Run dependent folders");

// Create and add the configuration folder to run
TFolder *configuration = run
  ->AddFolder("Configuration", "Run configuration");

// Create and add the run_mc folder
TFolder *run_mc = aliroot
  ->AddFolder("RunMC", "MonteCarlo run dependent folders");

// Create and add the configuration_mc folder to run_mc
TFolder *configuration_mc = run_mc
  ->AddFolder("Configuration", "MonteCarlo run configuration");
```

In this macro, the folder is also added to the list of browsables. This way, it is visible in the browser on the top level.

**Posting Data to a Folder (Producer)**

A `TFolder` can contain other folders as shown above or any `TObject` descendants. In general, users will not post a single object to a folder, they will store a collection or multiple collections in a folder. For example, to add an array to a folder:

```c++
TObjArray *array;
run_mc->Add(array);
```

**Reading Data from a Folder (Consumer)**

One can search for a folder or an object in a folder using the `TROOT::FindObjectAny` method. `FindObjectAny` analyzes the string passed as its argument and searches in the hierarchy until it finds an object or folder matching the name.

With `FindObjectAny`, you can give the full path name, or the name of the folder. If only the name of the folder is given, it will return the first instance of that name.

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

A string-based search is time consuming. If the retrieved object is used frequently or inside a loop, you should save a pointer to the object as a class data member. Use the naming service only in the initialization of the consumer class.

When a folder is deleted, any reference to it in the parent or other folder is deleted also.

By default, a folder does not own the object it contains. You can over- write that with `TFolder::SetOwner`. Once the folder is the owner of its contents, the contents are deleted when the folder is deleted.

Some `ROOT` objects are automatically added to the folder hierarchy. For example, the following folders exist on start up:

- `//root/ROOT Files` with the list of open Root files
- `//root/Classes` with the list of active classes
- `//root/Geometries` with active geometries
- `//root/Canvases` with the list of active canvases
- `//root/Styles` with the list of graphics styles
- `//root/Colors` with the list of active colors

For example, if a file `myFile.root` is added to the list of files, one can retrieve a pointer to the corresponding `TFile` object with a statement like:

```c++
TFile *myFile = (TFile*)gROOT->FindObjectAny("/ROOT Files/myFile.root");
// or
TFile *myFile = (TFile*)gROOT->FindObjectAny("myFile.root");
```

**Tasks**

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

An example of `TTask` subclasses is in `$ROOTSYS/tutorials/MyTasks.cxx`. An example script that creates a task hierarchy and adds it to the browser is `$ROOTSYS/tutorials/tasks.C`.
Here is part of `MyTasks.cxx` that shows how to subclass from `TTask`.

```c
#include "TTask.h"

class MyRun : public TTask {
public:
    MyRun() {};
    MyRun(const char *name, const char *title);
    virtual ~MyRun() {};
    void Exec(Option_t *option="");
    ClassDef(MyRun,1) // Run Reconstruction task
};

class MyEvent : public TTask {
public:
    MyEvent() {};
    MyEvent(const char *name, const char *title);
    virtual ~MyEvent() {};
    void Exec(Option_t *option="");
    ClassDef(MyEvent,1) // Event Reconstruction task
};
```

Later in `MyTasks.cxx`, we can see examples of the constructor and overridden `Exec` method:

```c
ClassImp(MyRun)
MyRun::MyRun(const char *name, const char *title) : TTask(name,title) {
    
    void MyRun::Exec(Option_t *option) {
        printf("MyRun executing\n");
    }
}
```

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

The constructor of `TTask` has two arguments: the name and the title. This script creates the task defined above, and creates a hierarchy of tasks.
Execute and Debug Tasks

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

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

A Task tree may be made persistent, saving the status of all the tasks.
11 Input/Output

This chapter covers the saving and reading of objects to and from ROOT files. It begins with an explanation of the physical layout of a ROOT file. It includes a discussion on compression, and file recovery. Then we explain the logical file, the class TFile and its methods. We show how to navigate in a file, how to save objects and read them back. We also include a discussion on Streamers. Streamers are the methods responsible to capture an objects current state to save it to disk or send it over the network. At the end of the chapter is a discussion on the two specialized ROOT files: TNetFile and TWebFile.

The Physical Layout of ROOT Files

A ROOT file is like a UNIX file directory. It can contain directories and objects organized in unlimited number of levels. It also is stored in machine independent format (ASCII, IEEE floating point, Big Endian byte ordering).

To look at the physical layout of a ROOT file, we first create one. This example creates a ROOT file and 15 histograms, fills each histogram with 1000 entries from a gaussian distribution, and writes them to the file.

```c
char name[10], title[20];
// Create an array of Histograms
TObjArray Hlist(0);
// create a pointer to a histogram
TH1F* h;
// make and fill 15 histograms
// and add them to the object array
for (Int_t i = 0; i < 15; i++) {
    sprintf(name,"h%d",i);
    sprintf(title,"histo nr:%d",i);
    h = new TH1F(name,title,100,-4,4);
    Hlist.Add(h);
    h->FillRandom("gaus",1000);
}
// open a file and write the array to the file
TFile f("demo.root","recreate");
Hlist->Write();
f.Close();
```

The example begins with a call to the TFile constructor. TFile is the class describing the ROOT file. In the next section, when we discuss the logical file structure, we will cover TFile in detail. You can also see that the file has the extension ".root", this convention is encouraged, however ROOT does not depend on it.

The last line of the example closed the file. To view its contents it needs to be opened again, and once opened we can view the contents in the ROOT Object browser by creating a TBrowser object.

```c
root [] TFile f("demo.root")
root [] TBrowser browser;
```

In the browser, we can see the 15 histograms we created.

![ROOT Object Browser](image)

Once we have the TFile object, we can call the TFile::Map() method to view the physical layout. The output of Map() prints the date/time, the start address of the record, the number of bytes in the record, the class name of the record, and the compression factor.

```c
root [] f.Map()
```

We see the fifteen histograms (TH1F's) with the first one starting at byte 148. We also see an entry TFile. You may notice that the first entry starts at byte 64. The first 64 bytes are taken by the file header.

```
root [] f.Map()
```
The File Header

This table shows the file header information:

<table>
<thead>
<tr>
<th>Byte</th>
<th>Value Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 -&gt; 4</td>
<td>root</td>
<td>Root file identifier</td>
</tr>
<tr>
<td>5 -&gt; 8</td>
<td>fVersion</td>
<td>File format version</td>
</tr>
<tr>
<td>9 -&gt; 12</td>
<td>fBEGIN</td>
<td>Pointer to first data record</td>
</tr>
<tr>
<td>13 -&gt; 16</td>
<td>fEND</td>
<td>Pointer to first free word at the EOF</td>
</tr>
<tr>
<td>17 -&gt; 20</td>
<td>fSeekFree</td>
<td>Pointer to FREE data record</td>
</tr>
<tr>
<td>25 -&gt; 28</td>
<td>nfrees</td>
<td>Number of free data records</td>
</tr>
<tr>
<td>33 -&gt; 37</td>
<td>fCompress</td>
<td>Zip compression level</td>
</tr>
</tbody>
</table>

The first four bytes of the file header contain the string "root" which identifies a file as a ROOT file. Because of this identifier, ROOT is not dependent on the ".root" extension. It is still a good idea to use the extension, just for us to recognize them easier.

The nfrees value is the number of free records. A ROOT file has a maximum size of 2 gigabytes. This variable along with fNBytesFree keeps track of the free space in terms of records and bytes. This count also includes the deleted records, which are available again.

The Top Directory Description

The 84 bytes after the file header contain the top directory description, including the name, the date and time it was created, and the date and time of the last modification.

The Histogram Records

What follows are the 15 histograms, in records of variable length.

<table>
<thead>
<tr>
<th>At: 64</th>
<th>20010404/092347</th>
<th>TFile</th>
</tr>
</thead>
</table>

The first four bytes of each record is an integer holding the number of bytes in this record. A negative number flags the record as deleted, and makes the space available for recycling in the next write. The rest of bytes in the header contain all the information to uniquely identify a data block on the file. This is followed by the object data.

This table explains the values in each individual record:

<table>
<thead>
<tr>
<th>Byte</th>
<th>Value Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 -&gt; 4</td>
<td>Nbytes</td>
<td>Length of compressed object (in bytes)</td>
</tr>
<tr>
<td>5 -&gt; 6</td>
<td>Version</td>
<td>TKey version identifier</td>
</tr>
<tr>
<td>7 -&gt; 10</td>
<td>ObjLen</td>
<td>Length of uncompressed object</td>
</tr>
<tr>
<td>11 -&gt; 14</td>
<td>Datime</td>
<td>Date and time when object was written to file</td>
</tr>
<tr>
<td>15 -&gt; 16</td>
<td>KeyLen</td>
<td>Length of the key structure (in bytes)</td>
</tr>
<tr>
<td>17 -&gt; 18</td>
<td>Cycle</td>
<td>Cycle of key</td>
</tr>
<tr>
<td>19 -&gt; 22</td>
<td>SeekKey</td>
<td>Pointer to record itself (consistency check)</td>
</tr>
<tr>
<td>23 -&gt; 26</td>
<td>SeekPdir</td>
<td>Pointer to directory header</td>
</tr>
<tr>
<td>27</td>
<td>lname</td>
<td>Number of bytes in the class name</td>
</tr>
<tr>
<td>28-&gt;..</td>
<td>ClassName</td>
<td>Object Class Name</td>
</tr>
<tr>
<td>..-&gt;..</td>
<td>Name</td>
<td>lName bytes with the name of the object</td>
</tr>
<tr>
<td>..-&gt;..</td>
<td>Title</td>
<td>Number of bytes in the object title</td>
</tr>
<tr>
<td>..-&gt;..</td>
<td>Title</td>
<td>Title of the object</td>
</tr>
</tbody>
</table>

---

You see a reference to TKey. It is explained in detail in the next section.

The Class Description List (StreamerInfo List)

The histogram records are followed by a list of class descriptions called StreamerInfo. The list contains the description of each class that has been written to file.

The class description is recursive, because to fully describe a class, its ancestors and object data members have to be described also. In demo.root, the class description list contains the description for:

- TH1F
- all classes in the TH1F inheritance tree
- all classes of the object data members
- all classes in the object data members' inheritance tree.

This description is implemented by the TStreamerInfo class, and is often referred to as simply StreamerInfo.
The record headers of objects in the file. For example, we can get the list of keys which is essentially an index to the objects in the file. The class, for example objects of old and new versions of a class can be in the same combination it encountered. The file can have multiple versions of the same version number.

ROOT allows a class to have multiple versions, and each version has its own description in form of a StreamerInfo. Above you see the class name and version number.

The StreamerInfo list has only one description for each class/version combination it encountered. The file can have multiple versions of the same class, for example objects of old and new versions of a class can be in the same file.

The StreamerInfo list is described in detail in the section on StreamerInfo.

The demo.root example contains only TH1F objects. Here we see the recursive nature of the class description, it contains the StreamerInfo of all the classes needed to describe TH1F.

StreamerInfo for class: TH1F, version=1
BASE TH1 offset= 0 type= 0 1-Dim histogram base class
BASE TH1F offset= 0 type= 0 Array of floats

StreamerInfo for class: TH1, version=3
BASE TH1 offset= 0 type=67 The basis for a named object (name, title)
BASE TAttLine offset= 0 type=0 Line attributes
BASE TAttFill offset= 0 type=0 Fill area attributes
BASE TAttMarker offset= 0 type=0 Marker attributes
Int_t fNcells offset= 0 type= 3 number of bins(1D), cells(2D) +U/Overflows
TAxis fXaxis offset= 0 type=61 X axis descriptor
TAxis fYaxis offset= 0 type=61 Y axis descriptor
TAxis fZaxis offset= 0 type=61 Z axis descriptor
Short_t fBarWidth offset= 0 type=62 Array of sum of squares of weights
Short_t fBarOffset offset= 0 type=62 Array to display contour levels
Short_t fBarWidth offset= 0 type=62 Array of sum of squares of weights
Stat_t fEntries offset= 0 type=62 Total Sum of weights
Stat_t fSumw offset= 0 type=62 Total Sum of squared of weights
Stat_t fSumw2 offset= 0 type=62 Total Sum of weight*X*X
Double_t fMean offset= 0 type=62 Mean value for plotting
Double_t fMinoffset offset= 0 type=62 Normalization factor
TArrayD fContour offset= 0 type=62 Array to display contour levels
TArrayD fContour2 offset= 0 type=62 Array of sum of squares of weights
TClass fFunctions offset= 0 type=62 Array of functions (file and user)

StreamerInfo for class: TNamed, version=1
- StreamerInfo for class: TAttLine, version=1
- StreamerInfo for class: TAttFill, version=1
- StreamerInfo for class: TAttMarker, version=1
- StreamerInfo for class: TAxis, version=6
- StreamerInfo for class: TKey, version=4

You can print a file's StreamerInfo list with the TFile::ShowStreamerInfo method. Below is an example of the output. Only the first line of each class description is shown.

The demo.root example contains only TH1F objects. Here we see the recursive nature of the class description, it contains the StreamerInfo of all the classes needed to describe TH1F.

You can print a file's StreamerInfo list with the TFile::ShowStreamerInfo method. Below is an example of the output. Only the first line of each class description is shown.

Warning in <TFile::TFile>: file demo.root probably not closed, trying to recover successfully recovered 15 keys

The Logical ROOT File: TFile and TKey

We saw that the TFile::Map() method reads the file sequentially and prints information about each record while scanning the file. It is not feasible to only support sequential access and hence ROOT provides random or direct access, i.e. reading a specified object at a time. To do so, TFile keeps a list of TKey, which is essentially an index to the objects in the file. The TKey class describes the record headers of objects in the file. For example, we can get the list of keys...
and print them. To find a specific object on the file we can use the `TFile::Get()` method.

```c
root [] TFile f("demo.root")
root [] f.GetListOfKeys()->Print()
TKey Name = h0, Title = histo nr:0, Cycle = 1
TKey Name = h1, Title = histo nr:1, Cycle = 1
TKey Name = h2, Title = histo nr:2, Cycle = 1
TKey Name = h3, Title = histo nr:3, Cycle = 1
TKey Name = h4, Title = histo nr:4, Cycle = 1
TKey Name = h5, Title = histo nr:5, Cycle = 1
TKey Name = h6, Title = histo nr:6, Cycle = 1
TKey Name = h7, Title = histo nr:7, Cycle = 1
TKey Name = h8, Title = histo nr:8, Cycle = 1
TKey Name = h9, Title = histo nr:9, Cycle = 1
TKey Name = h10, Title = histo nr:10, Cycle = 1
TKey Name = h11, Title = histo nr:11, Cycle = 1
TKey Name = h12, Title = histo nr:12, Cycle = 1
TKey Name = h13, Title = histo nr:13, Cycle = 1
TKey Name = h14, Title = histo nr:14, Cycle = 1
root [] TH1F *h9 = (TH1F*)f.Get("h9");
```

The `TFile::Get()` finds the TKey object with name "h9". Using the TKey info it will import in memory the object in the file at the file address #3352 (see the output from the `TFile::Map` above). This is done by the Streamer method that is covered in detail in a later section.

Since the keys are available in a TList of TKeys we can iterate over the list of keys:

```c
root [] .x iterate.C
key: h0 points to an object of class: TH1F at 150
key: h1 points to an object of class: TH1F at 503
key: h2 points to an object of class: TH1F at 854
key: h3 points to an object of class: TH1F at 1194
key: h4 points to an object of class: TH1F at 1539
key: h5 points to an object of class: TH1F at 1882
key: h6 points to an object of class: TH1F at 2240
key: h7 points to an object of class: TH1F at 2582
key: h8 points to an object of class: TH1F at 2937
key: h9 points to an object of class: TH1F at 3293
key: h10 points to an object of class: TH1F at 3639
key: h11 points to an object of class: TH1F at 3986
key: h12 points to an object of class: TH1F at 4339
key: h13 points to an object of class: TH1F at 4694
key: h14 points to an object of class: TH1F at 5038
```

In addition to the list of keys, `TFile` also keeps two other lists:

- `TFile::fFree` is a TList of free blocks used to recycle freed up space in the file. ROOT tries to find the best free block. If a free block matches the size of the new object to be stored, the object is written in the free block and this free block is deleted from the list. If not, the first free block bigger than the object is used.

- `TFile::fListHead` contains a sorted list (TSortedList) of objects in memory.

The diagram below illustrates the logical view of the `TFile` and `TKey`. 

![Logical view of TFile and TKey](image-url)
Viewing the Logical File Contents

**TFile** is a descendent of **TDirectory**, which means it behaves like a **TDirectory**. We can list the contents, print the name, and create subdirectories. In a ROOT session, you are always in a directory and the directory you are in is called the current directory and is stored in the global variable **gDirectory**.

Let's look at a more detailed example of a ROOT file and its role as the current directory. First, we create a ROOT file by executing a sample script.

```
root [] .x $ROOTSYS/tutorials/hsimple.C
```

Now you should have `hsimple.root` in your directory. The file was closed by the script so we have to open it again to work with it.

```
root [] TFile f (*hsimple.root", "UPDATE")
TFile** hsimple.root
TFile* 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
```

It shows the two lines starting with **TFile** followed by four lines starting with the word "KEY". The four keys tell us that there are four objects on disk in this file. The syntax of the listing is:

```
KEY: <class> <variable>;<cycle number> <title>
```

For example, the first line in the list means there is an object in the file on disk, called `hpx`. It is of the class `TH1F` (one-dimensional histogram of floating numbers). The object's title is "This is the px distribution".

If the line starts with **OBJ**, the object is in memory. The `<class>` is the name of the ROOT class (T-something). The `<variable>` is the name of the object. The cycle number along with the variable name uniquely identifies the object. The `<title>` is the string given in the constructor of the object as title.

The Current Directory

When you create a **TFile** object, it becomes the current directory. Therefore, the last file to be opened is always the current directory. To check your current directory you can type:

```
root [] gDirectory->pwd()
Rint:/
```

This means that the current directory is the ROOT session (`Rint`). When you create a file, and repeat the command the file becomes the current directory.

```
root [] TFile f1("AFile1.root");
root [] gDirectory->pwd()
AFile1.root:/
```

If you create two files, the last becomes the current directory.

```
root [] TFile f2("AFile2.root");
root [] gDirectory->pwd()
AFile2.root:/
```

To switch back to the first file, or to switch to any file in general, you can use the **TDirectory::cd** method. The next command changes the current directory back to the first file.

```
root [] f1.cd();
root [] gDirectory->pwd()
AFile1.root:/
```

Note that even if you open the file in "READ" mode, it still becomes the current directory.
CINT also offers a shortcut for `gDirectory->pwd()` and `gDirectory->ls()`, you can type:

```cpp
cint> .pwd
AFile1.root:
```

To return to the home directory, the one we were in before we opened any files:

```cpp
cint> gROOT->cd()
(unsigned char)1
cint> gROOT->pwd()
Rint:
```

### Objects in Memory and Objects on Disk

The `TFile::ls()` method has an option to list the objects on disk ("-d") or the objects in memory ("-m"). If no option is given it lists both, first the objects in memory, then the objects on disk. For example:

```cpp
cint> TFile *f = new TFile("hsimple.root");
cint> gDirectory->ls("-m")
TFile** hsimple.root
TFile* hsimple.root
OBJ: TProfile hprof Profile of pz versus px : 0
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: TProfile ntuple;1 Demo ntuple
```

Remember that `gDirectory` is the current directory and at this time is equivalent to `"f"`. This correctly states that no objects are in memory. The next command lists the objects on disk in the current directory.

```cpp
cint> gDirectory->ls("-d")
TFile** hsimple.root
TFile* 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
```

To bring an object from disk into memory, we have to use it or "Get" it explicitly. When we use the object, ROOT gets it for us. Any reference to `hprof` will read it from the file. For example drawing `hprof` will read it from the file and create an object in memory. Here we draw the profile histogram, and then we list the contents.

```cpp
root [] hprof->Draw()
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
cint> f->ls()
TFile** hsimple.root
TFile* hsimple.root
OBJ: TProfile hprof Profile of pz versus px : 0
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: TProfile ntuple;1 Demo ntuple
```

We now see a new line that starts with OBJ. This means that an object of class `TProfile`, called `hprof` has been added in memory to this directory. This new `hprof` in memory is independent from the `hprof` on disk. If we make changes to the `hprof` in memory, they are not propagated to the `hprof` on disk. A new version of `hprof` will be saved once we call `Write`.

You may wonder why `hprof` is added to the objects in the current directory. `hprof` is of the class `TProfile` that inherits from `TH1D`, which inherits from `TH1`. `TH1` is the basic histogram. All histograms and trees are created in the current directory (also see "Histograms and the Current Directory"). The reference to "all histograms" includes objects of any class descending directly or indirectly from `TH1`. Hence, our `TProfile` `hprof` is created in the current directory.

There was another side effect when we called the `TH1::Draw` method. CINT printed this statement:

```cpp
<TCanvas::MakeDefCanvas>: created default TCanvas with name c1
```

It tells us that a `TCanvas` was created and it named it `c1`. This is where ROOT is being nice, and it creates a canvas for drawing the histogram if no canvas was named in the draw command, and if no active canvas exists.

The newly created canvas, however, is NOT listed in the contents of the current directory. Why is that? The canvas is not added to the current directory, because by default ONLY histograms and trees are added to the object list of the current directory. Actually, `TEventList` objects are also added to the current directory, but at this time, we don't have to worry about those.

If the canvas is not in the current directory then where is it? Because it is a canvas, it was added to the list of canvases. This list can be obtained by the command `gROOT->GetListOfCanvases()->ls()`. The `ls()` will print the contents of the list. In our list, we have one canvas called `c1`. It has a `TFrame`, a `TProfile`, and a `TPaveStats`.

```cpp
root [] gROOT->GetListOfCanvases()->ls()
Canvas Name=c1 Title=c1 Option=
TCanvas f1lowEDEC0 f1lowEDEC0 fNEDC1 fNEDC1 fNEDC1 Name=c1 Title=c1 Option=
TFrame X1=-4.000000 Y1=0.000000 X2=4.000000 Y2=-19.384882
OBJ: TProfile hprof Profile of pz versus px : 0
TPaveText X1=-4.900000 Y1=20.475282 X2=-0.950000 Y2=21.686837 title
TPaveStats X1=2.800000 Y1=17.446395 X2=4.800000 Y2=21.323371 stats
```

Let's proceed with our example and draw one more histogram, and we see one more OBJ entry.
**Saving Histograms to Disk**

At this time, the objects in memory (OBJ) are identical to the objects on disk (KEY). Let's change that by adding a fill to the hpx we have in memory.

```c
root [] hpx->Fill(0)
```

Now the hpx in memory is different from the histogram (hpx) on disk.

Only one version of the object can be in memory, however, on disk we can store multiple versions of the object. The TFile::Write method will write the list of objects in the current directory to disk. It will add a new version of hpx and hprof.

```c
root [] f->Write()  
```

The TFile::Write method wrote the entire list of objects in the current directory to the file. You see that it added two new keys: hpx;2 and hprof;2 to the file. Unlike memory, a file is capable of storing multiple objects with the same name. Their cycle number, the number after the semicolon, differentiates objects on disk with the same name.

This picture shows the file before and after the call to Write.

If you wanted to save only hpx to the file, but not the entire list of objects, you could use the TH1::Write method of hpx:

```c
root [] hpx->Write()
```

A call to obj->Write without any parameters will call obj->GetName() to find the name of the object and use it to create a key with the same name. You can specify a new name by giving it as a parameter to the Write method.

```c
root [] hpx->Write("newName")
```

If you want to re-write the same object, with the same key, use the overwrite option.
If you give a new name and use the kOverwrite, the object on disk with the matching name is overwritten if such an object exists. If not, a new object with the new name will be created.

If you go along so far, you can see that c1 which was displaying hpx is now blank. Furthermore, hpx no longer exists.

This is important to remember, do not close the file until you are done with the objects or any attempt to reference the objects will fail.

In this case, you will need to do all the bookkeeping for all the created histograms.

**Saving Objects to Disk**

In addition to histograms and trees, you can save any object in a ROOT file. To save a canvas to the ROOT file you can use TDirectory::Write.

```
root [] f->Close()
root [] f->ls()
TFile** hsimple.root
TFile* hsimple.root

OBJ: TH1F hpx This is the px distribution
KEY: TH1F hpx;2 This is the px distribution
KEY: TH1F hpx;1 This is the px distribution
KEY: TH2F hpxpy;1 py vs px
KEY: TProfile hprof;2 Profile of pz versus px
KEY: TProfile hprof;1 Profile of pz versus px
KEY: TNtuple ntuple;1 Demo ntuple
KEY: TCanvas c1;1 c1
```

**Saving Collections to Disk**

All collection classes inherit from TCollection and hence inherit the TCollection::Write method. When you call TCollection::Write() each object in the container is written individually into its own key in the file.

To write all objects into one key you can specify the name of the key and use the TObject::kSingleKey option. For example:

```
root [] list = new TList;
root [] n1, n2;
root [] n1 = new TNamed("name1", "title1");
root [] n2 = new TNamed("name2", "title2");
root [] list->Add(n1);
root [] list->Add(n2);
root [] list->Write("list", TObject::kSingleKey);
```

**A TFile Object going Out of Scope**

There is another important point to remember about TFile::Close and TFile::Write. When a variable is declared on the stack in a function such as in the code below, it will be deleted when it goes out of scope.

```
void foo() {
    TFile f("AFile.root", "RECREATE");
}
```

As soon as the function foo has finished executing, the variable f is deleted. When a TFile object is deleted an implicit call to TFile::Close is made. This will save only the file descriptor to disk. It contains: the file header, the StreamerInfo list, the key list, the free segment list, and the end address (see
A `TFile::Close` does not make a call to `Write`, which means that the objects in memory will not be saved in the file.

You need to explicitly call `TFile::Write()` to save the object in memory to file before the exit of the function.

```cpp
void foo() {
  TFile f("AFile.root", "RECREATE");
  // stuff
  f->Write();
}
```

To prevent an object in a function from being deleted when it goes out of scope, you can create it on the heap instead of on the stack. This will create a `TFile` object `f`, that is available on a global scope, and it will still be available when exiting the function.

```cpp
void foo() {
  TFile *f = new TFile("AFile.root", "RECREATE");
}
```

### Retrieving Objects from Disk

If you have a ROOT session running, please quit and start fresh.

We saw that multiple versions of an object with the same name can be in a ROOT file. In our example, we saved a modified histogram `hpx` to the file, which resulted in two `hpx`'s uniquely identified by the cycle number: `hpx;1` and `hpx;2`. The question is how do we retrieve the right version of `hpx`.

When opening the file and using `hpx`, CINT retrieves the one with the highest cycle number.

To read the `hpx;1` into memory, rather than the `hpx;2` we would get by default, we have to explicitly get it and assign it to a variable.

```cpp
root [] TFile *f = new TFile("AFile.root", "RECREATE")
root [] f->mkdir("Wed011003")
(class TDirectory*)0x1072b5c8
root [] f->ls()
TFile**         AFile.root
TFile*         AFile.root
TDirectory*           Wed011003       Wed011003
KEY: TDirectory       Wed011003;1     Wed011003
```

To return to the file's top directory, use `cd()` without any arguments.

```cpp
root [] gFile->pwd()
AFile.root:/
```

### Subdirectories and Navigation

The `TDirectory` class lets you organize its contents into subdirectories, and `TFile` being a descendent of `TDirectory` inherits this ability.

Here is an example of a ROOT file with multiple subdirectories as seen in the ROOT browser.

**Creating Subdirectories**

To add a subdirectory to a file use `Directory::mkdir`.

```cpp
root [] TFile *f1 = new TFile("hsimple.root")
root [] TH1F *hpx1 = (TH1F*) f1->Get("hpx;1")
root [] hpx1->Draw()
```

The example below opens the file for writing and creates a subdirectory called "Wed011003". Listing the contents of the file shows the new directory in the file and the `TDirectory` object in memory.

```cpp
The example below opens the file for writing and creates a subdirectory called "Wed011003". Listing the contents of the file shows the new directory in the file and the `TDirectory` object in memory.

### Navigating to Subdirectories

We can change the current directory by navigating into the subdirectory, and after changing directory, we can see that `gDirectory` is now "Wed011003".

```cpp
root [] gDirectory->pwd()
AFile.root:/Wed011003
```

In addition to `gDirectory` we have `gFile`, another global that points to the current file.

In our example, `gDirectory` points to the subdirectory, and `gFile` points to the file (i.e. the files' top directory).

```cpp
root [] gFile->pwd()
AFile.root:/
```

To return to the file's top directory, use `cd()` without any arguments.

```cpp
root [] f->cd()
AFile.root:
```

Change to the subdirectory again, and create a histogram. It is added to the current directory, which is the subdirectory "Wed011003".

```cpp
root [] f->cd("Wed011003")
root [] TH1F histo = new TH1F("histo", histo", 10, 0, 10");
root [] gDirectory->ls()
TDirectory* Wed011003 Wed011003
OBJ: TH1F histo histo : 0
```

If you are in a subdirectory and you would like to have a pointer to the file containing the subdirectory, you can do so:

```cpp
root [] gDirectory->GetFile()
```

If you are in the top directory `gDirectory` is the same as `gFile`.

We write the file to save the histogram on disk, to show you how to retrieve it later.
When retrieving an object from a subdirectory, you can navigate to the subdirectory first or give it the path name relative to the file. The read object is created in memory in the current directory.

In this first example, we get `histo` from the top directory and the object will be in the top directory.

```c
TH1 *h = (TH1*) f->Get("Wed011003/histo;1")
```

If file is written, a copy of `histo` will be in the top directory. This is an effective way to copy an object from one directory to another.

In contrast, in the code box below, `histo` will be in memory in the subdirectory because we changed the current directory.

```c
f->cd("Wed011003");
TH1 *h = (TH1*) gDirectory->Get("histo;1")
```

Note that there is no warning if the retrieving was not successful. You need to explicitly check the value of `h`, and if it is null, the object could not be found. For example, if you did not give the path name the histogram cannot be found and the pointer to `h` is null:

```c
TH1 *h = (TH1*) gDirectory->Get("Wed011003/histo;1")
```

```c
h(class TH1*)0x10767de0
```

```c
TH1 *h = (TH1*) gDirectory->Get("histo;1")
```

```c
h(class TH1*)0x0
```

Removing Subdirectories

To remove a subdirectory you need to use `TDirectory::Delete`. There is no `TDirectory::rmdir`. The `Delete` method takes a string containing the variable name and cycle number as a parameter.

```c
void Delete(const char *namecycle)
```

The `namecycle` string has the format `name;cycle`. Here are some rules to remember:

- name = * means all, but don't remove the subdirectories
- cycle = * means all cycles (memory and file)
- cycle = ** means apply to a memory object
- cycle = 9999 also means apply to a memory object
- namecycle = ** means the same as namecycle = T**
- namecycle = T delete subdirectories

For example to delete a directory from a file, you must specify the directory cycle,

```c
f->Delete("Wed011003;1")
```

Some other examples of `namecycle` format are:

- foo: delete the object named foo from memory
- foo;1: delete the cycle 1 of the object named foo from the file
- foo;*: delete all cycles of foo from the file and also from memory
- **;2: delete all objects with cycle number 2 from the file
- **;*: delete all objects from memory and from the file
- T;*: delete all objects from memory and from the file including all subdirectories

Streamers

To follow the discussion on Streamers, you need to know what a simple data type is. A variable is of a simple data type if it cannot be decomposed into other types. Examples of simple data types are longs, shorts, floats, and chars. In contrast, a variable is of a composite data type if it can be decomposed. For example, classes, structures, and arrays are composite types. Simple types are also called primitive types, basic types, and CINT sometimes calls them fundamental types.

When we say, "writing an object to a file", we actually mean writing the current values of the data members. The most common way to do this is to decompose (also called the serialization of) the object into its data members and write them to disk. The decomposition is the job of the Streamer. Every class with ambitions to be stored in a file has a Streamer that decomposes it and "streams" its members into a buffer.

The methods of the class are not written to the file, it contains only the persistent data members.

To decompose the parent classes, the Streamer calls the Streamer of the parent classes. It moves up the inheritance tree until it reaches an ancestor without a parent.

To serialize the object data members it calls their Streamer. They in turn move up their own inheritance tree and so forth.

The simple data members are written to the buffer directly. Eventually the buffer contains all simple data members of all the classes that make up this particular object.

Streaming Pointers

An object pointer data member presents a challenge to the streaming software. If the object pointed to is saved every time it could create circular dependencies and consume large amounts of disk space. The network of references must be preserved on disk and recreated upon reading the file.

When ROOT encounters a pointer data member it calls the streamer of the object and labels it with a unique object identifier. The object identifier is unique for one I/O operation. If there is another reference to the object in the same I/O operation, the first object only referenced by its ID, it is not saved again.
When reading the file, the object is rebuilt and the references recalculated. In this way, the network of pointers and their objects is rebuilt and ready to use the same way it was used before it was persistent.

Automatically Generated Streamers

A Streamer usually calls other Streamers: the Streamer of its parents and data members. This architecture depends on all classes having Streamers, because eventually they will be called. To ensure that a class has a Streamer, `rootcint` automatically creates one in the `ClassDef` macro which is defined in `$ROOTSYS/include/Rtypes.h`. `ClassDef` defines several methods for any class, and one of them is the Streamer. The automatically generated Streamer is complete and can be used as long as no customization is needed.

The `Event` class is defined in `$ROOTSYS/test/Event.h`. Looking at the class definition, we find that it inherits from `TObject`. It is a simple example of a class with diverse data members.

```c++
class Event : public TObject {
    private:
        TDirectory    *fTransient;   /// current directory
        Float_t  fPt;         /// transient value
        char       fType[20];
        Int_t      fNtrack;
        Int_t      fNseg;
        Int_t      fNvertex;
        UInt_t     fFlag;
        Float_t    fTemperature;
        EventReader fEvtHdr;     /// don't split
        TClonesArray  *fTracks;   /// transients
        TH1F     *fH;             /// transient value
        Int_t      fMeasures[10];
        Float_t    fMatrix[4][4];
        Float_t    *fClosestDistance;  ///[fNvertex]
};
```

The `Event` class is added to the CINT dictionary by the `rootcint` utility. This is the `rootcint` statement in the `$ROOTSYS/test/Makefile`:

```bash
@rootcint -f EventDict.cxx -c Event.h EventLinkDef.h
```

The `EventDict.cxx` file contains the automatically generated Streamer for `Event`:

```c++
void Event::Streamer(TBuffer &R__b)
{
    // Stream an object of class Event.
    if (R__b.IsReading()) {
        Event::Class()->ReadBuffer(R__b, this);
    } else {
        Event::Class()->WriteBuffer(R__b, this);
    }
}
```

When writing an `Event` object, `TClass::WriteBuffer` is called. `WriteBuffer` writes the current version number of the `Event` class, and its contents into the buffer `R__b`.

The Streamer calls `TClass::ReadBuffer` when reading an `Event` object. The `ReadBuffer` method reads the information from buffer `R__b` into the `Event` object.

Transient Data Members (//!)

To prevent a data member from being written to the file, insert a "!" as the first character after the comment marks. For example, in this version of `Event`, the `fPt` and `fTransient` data members are not persistent.

```c++
class Event : public TObject {
    private:
        TDirectory    *fTransient;   /// current directory
        Float_t  fPt;         /// transient value
        ...;
};
```

The Pointer To Objects (//->)

The string "->" in the comment field of the members `*fH` and `*fTracks` instruct the automatic Streamer to assume these will point to valid objects and the Streamer of the objects can be called rather than the more expensive `R__b << fH`.

```c++
TClonesArray  *fTracks;     ///->
TH1F     *fH;             ///->
```

Variable Length Array

When the Streamer comes across a pointer to a simple type, it assumes it is an array. Somehow, it has to know how many elements are in the array to reserve enough space in the buffer and write out the appropriate number of elements. This is done in the class definition.

```c++
TClonesArray  *fTracks;     ///->
TH1F     *fH;             ///->
```
For example:

```cpp
class Event : public TObject {
  private:
    char           fType[20];
    Int_t          fNtrack;
    Int_t          fNseg;
    Int_t          fNvertex;
      
    Float_t       *fClosestDistance;  // [fNvertex]
};
```

The array `fClosestDistance` is defined as a pointer of floating point numbers. A comment mark (`//`), and the number in square brackets tell the Streamer the length of the array for this object. In general the syntax is:

```
<simple type>  *<name>  // [<length>]
```

The length cannot be an expression. If a variable is used, it needs to be an integer data member of the class. It must be defined ahead of its use, or in a base class.

**Prevent Splitting (||)**

If you want to prevent a data member from being split when writing it to a tree append the characters || right after the comment string. This only makes sense for object data members. For example:

```cpp
EventHeader  fEvtHdr;  // || do not split the header
```

**Streamers With Special Additions**

Most of the time you can let `rootcint` generate a Streamer for you. However, if you want to write your own Streamer you can do so.

For some classes, it may be necessary to execute some code before or after the read or write block in the automatic Streamer. For example after the execution of the read block, one can initialize some non persistent members. There are two reasons why you would need to write your own Streamer. If you have a complex STL container type data member that is not yet supported by ROOT, or if you have a non-persistent data member that you want to initialize to a value depending on the read data members. In addition, the automatic Streamer does not support C-structures. It is best to convert the structure to a class definition.

First, you need to tell `rootcint` not to build a Streamer for you. The input to the `rootcint` command (in the makefile) is a list of classes in a `LinkDef.h` file. For example, the list of classes for `Event` are listed in `$ROOTSYS/test/EventLinkDef.h`. The "-" at the end of the class name tells `rootcint` not to generate a Streamer. In the example, you can see the `Event` class is the only one for which `rootcint` is instructed not to generate a Streamer.

```cpp
#pragma link C++ class EventHeader-;
```

The "-" sign tells `rootcint` to use the new Streamer system introduced in ROOT 3.0.

This is an example of a customized Streamer for `Event`:

```cpp
void Event::Streamer(TBuffer &R__b) {
  if (R__b.IsReading()) {
    Event::Class()->ReadBuffer(R__b, this);
    fTransient = gDirectory;  // save current directory
    fPt= TMath::Sqrt(fPx*fPx + fPy*fPy + fPz*fPz);
  } else {
    Event::Class()->WriteBuffer(R__b, this);
  }
}
```

**Writing Objects**

The Streamer decomposes the objects into data members and writes them to a buffer. It does not write the buffer to a file, it simply populates a buffer with bytes representing the object. This allows us to write the buffer to a file or do anything else we could do with the buffer. ... we can write it to a socket to send it over the network. This is beyond the scope of this chapter, but it is worthwhile to emphasize the need and advantage of separating the creation of the buffer from its use. Let's look how a buffer is written to a file.

A class needs to inherit from `TObject` or use `TDirectory::Write(obj)` to be saved to disk. However, a class that is a data member of another class does not have to inherit from `TObject`, it only has to have a Streamer. `EventHeader` is an example of such a case.

The `TObject::Write` method does the following:

1. Creates a `TKey` object in the current directory
2. Creates a `TBuffer` object which is part of the newly created `TKey`
3. Fills the `TBuffer` with a call to the class::Streamer method
4. Creates a second buffer for compression, if needed
5. Reserves space by scanning the `TFree` list. At this point, the size of the buffer is known.
6. Writes the buffer to the file
7. Releases the `TBuffer` part of the key
In other words, the TObject::Write calls the Streamer method of the class to build the buffer. The buffer is in the key and the key is written to disk. Once written to disk the memory consumed by the buffer part is released. The key part of the TKey is kept. The key consumes about 60 bytes, where the buffer since it contains the object data can be very large.

This is a diagram of a streamed TH1F in the buffer:

Ignore Object Streamers
You can instruct your class to ignore the TObject Streamer with the MyClass::Class::IgnoreTObjectStreamer method. When the class kIgnoreTObjectStreamer bit is set (by calling the IgnoreTObjectStreamer method), the automatically generated Streamer will not call TObject::Streamer, and the TObject part of the class is not streamed to the file. This is useful in case you do not use the TObject fBits and fUniqueID data members. You gain space on the file, and you do not lose functionality if you do not use the fBits and fUniqueID (see the section on TObject on the use of fBits and fUniqueID).

Streaming a TClonesArray
When writing a TClonesArray it bypasses by default the Streamer of the member class and uses a more efficient internal mechanism to write the members to the file.

You can override the default and specify that the member class Streamer is used by setting the TClonesArray::BypassStreamer bit to false:

```cpp
TClonesArray *fTracks;
fTracks->BypassStreamer(kFALSE); // use the member Streamer
```

When the kBypassStreamer bit is set, the automatically generated Streamer can call TClass::WriteBuffer directly. Bypassing the Streamer improves the performance when writing/reading the objects in the TClonesArray. However, the drawback is when a TClonesArray is written with split=0 bypassing the Streamer, the StreamerInfo of the class in the array being optimized, one cannot later use the TClonesArray with split>0.

For example, there is a problem with the following scenario:

1. a class Foo has a TClonesArray of Bar objects
2. the Foo object is written with split=0 to Tree T1. In this case the StreamerInfo for the class Bar is created in optimized mode by such a way that data members of the same type are written as an array improving the I/O performance.
3. in a new program, T1 is read and a new Tree T2 is created with the object Foo in split>1.

When the T2 branch is created, the StreamerInfo for the class Bar is created with no optimization (mandatory for the split mode). The optimized Bar StreamerInfo is going to be used to read the TClonesArray in T1. The result will be Bar objects with data member values not in the right sequence. The solution to this problem is to call BypassStreamer(kFALSE) for the TClonesArray. In this case, the normal Bar::Streamer function will be called. The BAR::Streamer function works OK independently if the Bar StreamerInfo had been generated in optimized mode or not.

Pointers and References in Persistency
An object pointer data member presents a challenge to the streaming software. If the object pointed to is saved every time, it could create circular dependencies and consume a large amount of disk space. The network of references must be preserved on disk and recreated upon reading the file.

If you use independent I/O operations for pointers and their referenced object you can use the TRef class. Later in this section is an example that compares disk space, memory usage, and I/O times of C++ pointers and TRefs. In general, a TRef is faster than C++ but the advantage of a C++ pointer is that it is already C++.

Streaming C++ Pointers
When ROOT encounters a pointer data member it calls the Streamer of the object and labels it with a unique object identifier. The object identifier is unique for one I/O operation. If there is another pointer to the object in the same I/O operation, the first object is referenced i.e. it is not saved again. When reading the file, the object is rebuilt and the references recalculated. In this way, the network of pointers and their objects is rebuilt and ready to use the same way it was used before it was persistent.
write the referenced objects, and multiple copies will exist. In addition, if the pointer is read before the referenced object, it is null and may cause a run time system error.

To address these limitations, ROOT offers the TRef class. TRef allows referencing an object in a different branch and/or in a different file. TRef also supports the complex situation where a TFile is updated multiple times on the same machine or a different machine.

When a TRef is read before its referenced object, it is null. As soon as the referenced object is read, the TRef points to it. In addition, one can specify an action to be taken by TRef in the case it is read before its reference object (see Action on Demand below).

Using TRef

A TRef is a lightweight object pointing to any TObject. This object can be used instead of normal C++ pointers in case

- The referenced object R and the pointer P are not written to the same file
- P is read before R
- R and P are written to different Tree branches

Below is a line from the example in $ROOTSYS/test/Event.cxx.

```c
TRef fLastTrack; //pointer to last track
```

```c
Track *track = new(Track::Random) Track();
//Save reference to last Track in the collection of Tracks
fLastTrack = track;
```

The track and its reference fLastTrack can be written with two separate iO calls in the same or in different files, in the same or in different branches of a TTree.

If the TRef is read and the referenced object has not yet been read, TRef will return a null pointer. As soon as the referenced object will be read, TRef will point to it.

How does it work?

A TRef is itself a TObject with an additional transient pointer fPID. When a TRef is used to point to a TObject *R.

For example in a class with

```c
TRef P;
```

one can do:

```c
P = R;  //to set the pointer
```

When the statement P = R is executed, the following happens:

- The pointer fPID is set to the current TProcessID (see below).
- The current ObjectNumber (see below) is incremented by one.
- R.fUniqueID is set to ObjectNumber.
- In the fPID object, the element fObjects[ObjectNumber] is set to P
- P.fUniqueID is also set to ObjectNumber.

After having set P, one can immediately return the value of R using P.GetObject(). This function returns the fObjects[fUniqueID] from the fPID object.

When the TRef is written, the process id number pidf of fPID is written in addition to the TObject part of TRef (fBits,fUniqueID).

When the TRef is read, its pointer fPID is set to the value stored in the TObject of TFile::fProcessID (fProcessIDs[pidf]).

When a referenced object is written, TObject::Streamer writes the pidf in addition to the standard fBits and fUniqueID.

When TObject::Streamer reads a reference object, the pidf is read. At this point, the referenced object is entered into the table of objects of the TProcessID corresponding to pidf.

WARNING: If MyClass is the class of the referenced object, The TObject part of MyClass must be streamed. One should not call MyClass::Class()->IgnoreTObjectStreamer()

TProcessID and TUUID

A TProcessID uniquely identifies a ROOT job. The TProcessID file consists of a TUID object, which provides a globally unique identifier.

The TUID class implements the UUID (Universally Unique Identifier), also known as GUID (Globally Unique Identifier). A UUID is 128 bits long, and if generated according to this algorithm, is either guaranteed to be different from all other UUID generated until 3400 A.D. or extremely likely to be different.

The TROOT constructor automatically creates a TProcessID. When a TFile contains referenced objects, the TProcessID object is written to the file. If a file has been written in multiple sessions (same machine or not), a TProcessID is written for each session. The TProcessID objects are used by TRef to uniquely identify the referenced TObject.

When a referenced object is read from a file (its bit fIsReferenced is set), this object is entered into the objects table of the corresponding TProcessID. Each TFile has a list of TProcessIDs (see TFile::fProcessIDs) also accessible via TProcessID::fgPIDs (for all files).

When this object is deleted, it is removed from the table via the cleanup mechanism invoked by the TObject destructor.

Each TProcessID has a table (TObject *fObjects) that keeps track of all referenced objects. If a referenced object has a UniqueID, a pointer to this unique object may be found via fObjects->At(fUniqueID). In the same way, when a TRef::GetObject is called, GetObject uses its own fUniqueID to find the pointer to the referenced object. See TProcessID::GetObjectWithID and PutObjectWithID.

ObjectNumber

When an object is referenced, a unique identifier is computed and stored in both the fUniqueID of the referenced and referencing object. This uniqueID is computed by incrementing by one the static global in TProcessID::fgNumber. fUniqueID is some sort of serial object number in the current session. One can retrieve at any time the current value of fgNumber by calling the static function TProcessID::GetObjectCount or set this number via TProcessID::SetObjectCount. To avoid a growing table of fObjects in TProcessID, in case, for example, one processes many events in a loop, it
might be necessary to reset the ObjectNumber at the end of processing of one event. See an example in $ROOTSYS/test/Event.cxx (look at function Build).

The value of ObjectNumber may be saved at the beginning of one event and reset to this original value at the end of the event. These actions may be nested.

```cpp
saveNumber = TProcessID::GetObjectCount();
TProcessID::SetObjectCount(savedNumber);
```

### Action on Demand

The normal behavior of a TRef has been described above. In addition, TRef supports "Actions on Demand". It may happen that the object referenced is not yet in memory, on a separate file or not yet computed. In this case, TRef is able to automatically execute an action:

- Call to a compiled function (static function of member function)
- Call to an interpreted function
- Execution of a CINT script

### How to select this option?

In the definition of the TRef data member in the original class, do:

```cpp
TRef fRef; //EXEC:execName points to something
```

When the special keyword "EXEC:*" is found in the comment field of the member, the next string is assumed to be the name of a TExec object. When a file is connected, the dictionary of the classes on the file is read in memory (see TFile::ReadStreamerInfo). When the TStreamerElement object is read, a TExec object is automatically created with the name specified after the keyword "EXEC:*". In case a TExec with a same name does not already exist.

The action to be executed via this TExec can be specified with:

- A call to the TExec constructor, if the constructor is called before
- Opening the file.
- A call to TExec::SetAction at any time.

One can compute a pointer to an existing TExec with a name with:

```cpp
TExec *myExec = gROOT->GetExec(execName);
myExec->SetAction(actionCommand);
```

actionCommand is a string containing a CINT instruction.

Examples:

```cpp
myExec->SetAction("LoadHits()");
myExec->SetAction(".x script.C");
```

When a TRef is de-referenced via TRef::GetObject, its TExec is automatically executed. The TExec function/script can do one or more of the following:

- Load a file containing the referenced object. This function typically looks in the file catalog (GRID).
- Compute a pointer to the referenced object and communicate this pointer back to the calling function TRef::GetObject via:

```cpp
TRef::SetObject(object).
```

As soon as an object is returned to GetObject, the fUniqueID of the TRef is set to the fUniqueID of the referenced object. At the next call to GetObject, the pointer stored in fPID:fObjects[fUniqueID] will be returned directly.

An example of action on demand is in $ROOTSYS/test/Event.h.

```cpp
TRef fWebHistogram; //EXEC:GetWebHistogram
```

When calling fWebHistogram.GetObject(), the function GetObject will automatically invoke the script GetWebHistogram.C via the interpreter. An example of a GetWebHistogram.C script is shown below:

```cpp
void GetWebHistogram() {
  TFile *f=TFile::Open("http://root.cern.ch/files/pippa.root");
  f->cd("DM/CJ");
  TH1 *h6 = (TH1*)gDirectory->Get("h6");
  h6->SetDirectory(0); //one should always do that
  delete f;
  TRef::SetObject(h6);
}
```

In the above example, a call to fWebHistogram.GetObject() executes the script with the function GetWebHistogram. This script connects to a file with histograms: pippa.root on the ROOT Web site and returns the object h6 to TRef::GetObject.

Note that if the definition of the TRef fWebHistogram had been:

```cpp
TRef fWebHistogram; //EXEC:GetWebHistogram()
```

The compiled or interpreted function GetWebHistogram() would have been called instead of the CINT script GetWebHistogram.C.

### Array of TRef

When storing multiple TRef's, it is more efficient to use a TRefArray. The efficiency is due to having a single pointer forID for all TRefs in the array. It has a dynamic compact table of fUniqueIDs. We recommend that you use a TRefArray rather than a collection of TRefs.

**Example:**

- Suppose a TObject *mytracks containing a list of Track objects.
- Suppose a TRefArray *pions containing pointers to the pion tracks in mytracks. This list is created with statements like: pions->Add(track);
- Suppose a TRefArray *muons containing pointers to the muon tracks in mytracks.

The 3 arrays mytracks, pions and muons may be written separately.
Schema Evolution

Schema evolution is a problem faced by long-lived data. When a schema changes, existing persistent data can become inaccessible unless the system provides a mechanism to access data created with previous versions of the schema.

In the lifetime of a collaboration, the class definitions (i.e., the schema) are likely to change frequently. Not only can the class itself change, but any of its parent classes or data member classes can change also. This makes the support for schema evolution necessary.

ROOT fully supports schema evolution. The diagram below illustrates some of the scenarios.

The top half represents different versions of the shared library with the class definitions. These are the in-memory class versions.

The bottom half represents data files that contain different versions of the classes.

1) An old version of a shared library and a file with new class definitions. This can be the case when someone has not updated the library and is reading a new file.
2) Reading a file with a shared library that is missing a class definition (i.e., missing class D).
3) Reading a file without any class definitions. This can be the case where the class definition is lost, or unavailable.
4) The current version of a shared library and an old file with old class versions (backward compatibility). This is often the case when reading old data.
5) Reading a file with a shared library built with MakeProject. This is the case when someone has already read the data without a shared library and has used ROOT's MakeProject feature to reconstruct the class definitions and shared library (MakeProject is explained in detail later on).

In case of a mismatch between the in-memory version and the persistent version of a class, ROOT maps the persistent one to the one in memory. This allows you to change the class definition at will, for example:

1) Change the order of data members in the class.
2) Add new data members. By default the value of the missing member will be 0 or in case of an object it will be set to null.
3) Remove data members.
4) Move a data member to a base class or vice versa.
5) Change the type of a member if it is a simple type or a pointer to a simple type. If a loss of precision occurs, a warning is given.
6) Add or remove a base class

In case of a mismatch between the in-memory version and the persistent version of a class, ROOT maps the persistent one to the one in memory.
The StreamerInfoElement Class

A StreamerInfoElement describes a data member of a simple type, object, array, pointer, or container. The offset in the StreamerInfoElement is the starting address of the data for that data member.

```c++
enum EReadWrite {
  kRead = 0, kWrite = 1, kReadWrite = 2
};

class StreamerInfoElement {
  public:
    StreamerInfoElement(const char* name, TData type, int offset);
    ~StreamerInfoElement();
   
    TClass* fClass;
    const char* fName;
    TData fType;
    int fOffset;
    int fLen;
    int fMethod;
   
    StreamerInfoElement* fNext;
    StreamerInfoElement* fPrevious;
  
   
    static void* GetFromObject(void* ptr, EReadWrite r); 
    static StreamerInfoElement* GetElement(void* ptr, TClass* class, int offset);
};
```

### Example: TH1 StreamerInfo

In the StreamerInfo of the TH1 class we see the four base classes: TNamed, TAttLine, TAttFill, and TAttMarker. These are followed by a list of the data members. Each data member is implemented by a StreamerInfoElement.

```c++
optimizedStreamerInfo = fStreamerInfo->ls();
optimizedStreamerInfo = getClassDef(TH1, 3);
```

### Optimized StreamerInfo

The entries starting with `i = 0` is the optimized format of the StreamerInfo. Consecutive data members of the same simple type and size are collapsed and read at once into an array for performance optimization.

```c++
for (i = 0; i < fEntries; i++) {
    double fNormFactor = fNormFactor[i];
    double fMinimum = fMinimum[i];
    double fMaximum = fMaximum[i];
    // other data members...
}
```

### Automatic Schema Evolution

When a class is defined in ROOT, it must include the ClassDef macro as the last line in the header file inside the class definition. The syntax is:

```c++
ClassDef (<ClassName>, <VersionNumber>)
```

The version number identifies this particular version of the class. The version number is written to the file in the Streamer by the call TBuffer::WriteVersion. You, as the designer of the class, do not need to do any manual modification in the Streamer. ROOT's schema evolution mechanism is automatic and handled by the StreamerInfo.
Manual Schema Evolution

If you have written your own Streamer as described in the section “Streamers With Special Additions”, you will have to manually add code for each version and manage the evolution of your class.

When you add or remove data members, you must modify the Streamer by hand. ROOT assumes that you have increased the class version number in the ClassDef statement and introduced the relevant test in the read part of the Streamer.

For example, if a new version of the Event class above includes a new member: int New the ClassDef statement should be changed to:

```
ClassDef(Event,2)
```

and the following lines should be added to the read part of the Streamer:

```c
if (R__v > 1) {
    f_New = R__b >> fNew;
} else {
    fNew = 0; // set to some default value
}
```

If, in the same new version 2 you remove the member fil, you must add the following code to read the histogram object into some temporary object and delete it:

```c
if (R__v < 2) {
    TH1F *dummy = 0;
    R__b >> dummy; delete dummy;
}
```

Our experience with manual schema evolution shows that it is easy to make and mismatches between Streamer writers and readers are frequent and increase as the number of classes increases.

We recommend you use rootcint generated Streamers whenever you can, and profit from the automatic schema evolution.

Building Class Definitions With The StreamerInfo

A ROOT file's streamerInfo list contains the description of all versions of all classes in the file. When a file is opened the StreamerInfo is read into memory and it provides enough information to make the file browseable.

The StreamerInfo enables us to recreate a header file for the class in case the compiled class is not available. This is done with the TFile::MakeProject method. It creates a directory with the header files for the named classes and a makefile to compile a shared library with the class definitions.

```c
// Build a 'fake' class
root [] gROOT->GetClass("ATLMuon")
(const class TClass*)0x87e5c08
```

We can create a Class using gROOT->GetClass, which makes a fake class from the StreamerInfo.

```c
// The fake class has a StreamerInfo
root [] gROOT->GetClass("ATLMuon")->GetStreamerInfo()->_is();
```

When we open the file, we get a warning that the file contains classes that are not in the CINT dictionary. This is correct since we did not load the class definitions.

```
Warning in <TClass::TClass>: no dictionary for class ATLMuon is available
```

However, when we try to use a specific class, we get a warning because the class is not in the CINT dictionary.

```
Warning in <TClass::TClass>: no dictionary for class TMCParticle is available
```

We can see the StreamerInfo for the classes:

```
root[] f.ShowStreamerInfo()
```

```
StreamerInfo for class: ATLMuon, version:1
BASE TObject offset= 0 type=66 Basic ROOT object
BASE TAtt3D offset= 0 type= 3D attributes
Int_t m_EPcode offset= 0 type= 3 Muon EP-code
Int_t m_MCParticle offset= 20 type= 3 Muon position in MCParticles list
Int_t m_KFmother offset= 24 type= 3 Muon mother KF-code
Int_t m_MCParticle offset= 20 type= 3 Muon position in MCParticles list
Int_t m_Eta offset= 36 type= 5 Eta coordinate
Int_t m_Phi offset= 40 type= 5 Phi coordinate
Int_t m_MCParticle offset= 20 type= 3 Muon position in MCParticles list
Int_t m_KFcode offset= 16 type= 3 Muon KF-code
Int_t m_Isolated offset= 32 type= 3 Muon isolation (1 for isolated)
Int_t m_UseFlag offset= 28 type= 3 Muon energy usage flag
Int_t m_MCParticle offset= 20 type= 3 Muon position in MCParticles list
Int_t m_EPcode offset= 16 type= 3 Muon EP-code
Int_t m_KFmother offset= 24 type= 3 Muon mother KF-code
Int_t m_Isolated offset= 32 type= 3 Muon isolation
Int_t m_Eta offset= 36 type= 5 Eta coordinate
Int_t m_Phi offset= 40 type= 5 Phi coordinate
Float_t m_PT offset= 44 type= 5 Transverse energy
Int_t m_Trigger offset= 0 type= 3 Result of trigger
```

Example: MakeProject

To explain the details, we use the example of the ATLFast project which is a fast simulation for the ATLAS experiment. The complete source for ATLFast can be downloaded at: http://root.cern.ch/root/atlfast.tar.gz.

Once we compile and run ATLFast we get a ROOT file called atlfast.root, containing the ATLFast objects.

```
MakeProject has three parameters:

MakeProject(const char *dirname, const char *classes, Option_t *option)
```

The first is the directory name in which to place the generated header files.

The second parameter is the name of the classes to include in the project. By default all classes are included. It recognizes the wild card character *, for example: "ATLF**" includes all classes beginning with ATLF.
The third parameter is an option with the following values:

- "new": If the directory does not exist, it is created.
- "recreate": If the directory does not exist, it creates as in "new", in addition if the directory does exist, all existing files are deleted before creating the new files.
- "update": The new classes are added to the existing directory and the existing classes are replaced with the new definition. If the directory does not exist, it creates it as in "new".
- "+": This option can be used in combination with the other three. It will create the necessary files to easily build a shared library containing the class definitions. Specifically it will:
  - Generate a script called `MAKE` that builds the shared library containing the definition of all classes in the directory.
  - Generate a `LinkDef.h` files to use with `rootcint` in `MAKE`.
  - Run `rootcint` to generate a `<dirname>ProjectDict.cxx` file with the current options in `compiledata.h`.
  - Build a shared library `<dirname>.so`

- "+*: This option can be used instead of the single "+". It does everything the single "+" does, and dynamically loads the shared library `<dirname>.so`.

This example makes a directory called `MyProject` that will contain all class definition from the `atlfast.root` file. The necessary makefile to build a shared library are also created, and since the "+*" is appended, the shared library is also loaded.

```
root [] f.MakeProject("MyProject","***","recreate++;")
MakeProject has generated 0 classes in MyProject
MyProject/MAKE file has been generated
Shared lib MyProject/MyProject.so has been generated
Shared lib MyProject/MyProject.so has been dynamically linked
```

The contents of `MyProject`:
```
ATLFCluster.h  ATLFPJet.h  ATLFPiscMaker.h
ATLFPTrack.h   MAKE     TMCParticle.h
ATLFPClusterMaker.h ATLFPJetMaker.h ATLFPmuon.h
ATLFPTrackMaker.h MyProject.so
ATLFPElectron.h ATLFPMCMaker.h ATLFPmuonMaker.h
ATLFPTrigger.h  MyProjectProjectDict.cxx
ATLFPElectronMaker.h ATLFPmaker.h ATLFPPhoton.h
ATLFP TriggerMaker.h MyProjectProjectDict.h
ATLFPHistBrowser.h ATLFPisc.h  ATLFPPhotonMaker.h
LinkDef.h       MyProjectProjectDict.o
```

Now you can load the shared library in any consecutive root session to use the `atlfast` classes.
```
root [] gSystem->Load("MyProject/MyProject")
root [] ATLFPmuon muon
```

This is an example of a generated header file:
```
/////////////////////////////////////////////////////////
//   This class has been generated by TFile::MakeProject
//     (Thu Apr  5 10:18:37 2001 by ROOT version 3.00/06)
//      from the StreamerInfo in file atlfast.root
/////////////////////////////////////////////////////////

#ifndef ATLFMuon_h
#define ATLFMuon_h
#include "TObject.h"
#include "TAtt3D.h"

class ATLFMuon : public TObject, public TAtt3D {
public:
    Int_t       m_KFcode;    //Muon KF-code
    Int_t       m_MCParticle; //Muon position in MCParticles list
    Int_t       m_KFmother;  //Muon mother KF-code
    Int_t       m_UseFlag;   //Muon energy usage flag
    Int_t       m_Isolated;  //Muon isolation (1 for isolated)
    Float_t     m_Eta;       //Eta coordinate
    Float_t     m_Phi;       //Phi coordinate
    Float_t     m_PT;        //Transverse energy
    Int_t       m_Trigger;   //Result of trigger

    ATLFMuon() { ; }
    virtual ~ATLFMuon() { ; }
    ClassDef(ATLFMuon,1) //
};
ClassImp(ATLFMuon)
#endif
```
Migrating to ROOT 3

We will distinguish the following cases:

**Case A**: You have your own Streamer method in your class implementation file. This also means that you have specified MyClass in the LinkDef.h file.

- Keep MyClass unchanged.
- Increment your class version id in ClassDef by 1, e.g. ClassDef(MyClass, 2).
- Change your Streamer function in the following way: The old write block can be replaced by the new standard Write. Change the read block to use the new scheme for the new versions and the old code for the old versions.

```cpp
void MyClass::Streamer(TBuffer &R__b)
{
    // Stream an object of class MyClass.
    if (R__b.IsReading()) {
        UInt_t R__s, R__c;
        Version_t R__v = R__b.ReadVersion(&R__s, &R__c);
        if (R__v > 1) {
            MyClass::Class()->ReadBuffer(R__b, this, R__v, R__s, R__c);
            return;
        }
        // process old versions before automatic schema evolution
        R__b >> xxxx;
        R__b >> .. etc
        R__b.CheckByteCount(R__s, R__c, MyClass::IsA());
        // end of old versions
    } else {
        MyClass::Class()->WriteBuffer(R__b, this);
    }
}
```

**Case B**: You use the automatic streamer in the dictionary file.

- Move the old Streamer from the file generated by rootcint to your class implementation file, then modify the Streamer function as in Case A above.
- Increment your class version id in ClassDef by 1, for example ClassDef(MyClass, 2).
- Add option "-" in the pragma line of LinkDef.

**Case C**: You use the automatic streamer in the dictionary file and you already use the option "+" in the LinkDef file. If the old automatic Streamer does not contain any statement using the function WriteArray, you have nothing to do, except running rootcint again to regenerate the new form of the Streamer function, otherwise proceed like for case B.

Compression and Performance

ROOT uses a compression algorithm based on the well-known gzip algorithm. It supports nine levels of compression. The default for ROOT is one.

The compression level can be set with the method TFile::SetCompressionLevel. Experience with this algorithm shows that a compression level of 1.3 for raw data files and around two on most DST files is the optimum. The choice of one for the default is a compromise between the time it takes to read and write the object vs. the disk space savings.

To specify no compression, set the level to zero.

We recommend using compression when the time spent in I/O is small compared to the total processing time. If the I/O operation is increased by a factor of 5 it is still a small percentage of the total time and it may compress the data by a factor of 10. On the other hand if the time spent on I/O is large, compression may have a large impact on the programs performance.

The compression factor, i.e. the savings of disk space, varies with the type of data. A buffer with a same value array is compressed so that the value is only written once. For example a track has the mass of a pion which is always the same, and the charge of the pion which is either positive or negative. For 1000 pions, the mass will be written only once, and the charge only twice (positive and negative).

When the data is sparse, i.e. when there are many zeros, the compression factor is also high.

The time to uncompress an object is small compared to the compression time and is independent of the selected compression level. Note that the compression level may be changed at any time, but the new compression level will only apply to newly written objects. Consequently, a ROOT file may contain objects with different compression levels.

The following table shows four runs of the demo script that creates 15 histograms with different compression parameters. To make the numbers more significant, the macro was modified to create 1000 histograms.

<table>
<thead>
<tr>
<th>Compression level</th>
<th>Bytes</th>
<th>Write Time</th>
<th>Read Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1,004,998</td>
<td>4.77</td>
<td>0.07</td>
</tr>
<tr>
<td>1</td>
<td>438,366</td>
<td>6.67</td>
<td>0.05</td>
</tr>
<tr>
<td>5</td>
<td>429,871</td>
<td>7.03</td>
<td>0.06</td>
</tr>
<tr>
<td>9</td>
<td>426,899</td>
<td>8.47</td>
<td>0.05</td>
</tr>
</tbody>
</table>

We have included two more examples to show the impact of compression on Trees in the next chapter.
Accessing ROOT Files Remotely via a rootd

Reading and writing ROOT files 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 behavior. 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.cern.ch: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 (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 1094 and this default port is assumed by TNetFile (actually by TUrl which is used by TNetFile). The port number has been allocated by the IANA and is reserved for ROOT.

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()
   - 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.

A Simple Session

In the above example, rootd runs on the remote node under user id minuser and listens to port 1094. When creating a TNetFile object you have to specify the same port number 1094 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 serve 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.

The rootd Daemon

The rootd daemon works with the TNetFile class. It allows remote access to ROOT database files in 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 (1094) on which your private rootd will listen. Using TNetFile you can now read and write files on the remote machine.

For example:

hpsalo [] telnet fsgi02.fnal.gov
login: minuser
Password: <fsgi02> rootd -p 1094
<fgs102> exit
hpsalo [] root
root [] TFile *f = TFile::Open("/root.cern.ch/~rdm/hsimple.root")
Name (pcna49a:rdm):
Password:
root [] f.ls()
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 privileges to do so). Add to /etc/services the line:

```
rootd 1094/tcp
```

to /etc/inetd.conf the line:

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

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

When 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 1094 as specified above in the /etc/services file.

Command Line Arguments for rootd

rootd supports the following arguments:

-\texttt{-i} \quad \text{says we are started by inetd}
-\texttt{-p \#} \quad \text{specifies port number to listen on}
-\texttt{-d \#} \quad \text{level 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 ROOT specific module to your Apache web server, you can distribute ROOT files to any ROOT user. There is no longer a need to send your files via FTP and risking (out of date) histograms or other objects. Your latest up-to-date results are always accessible to all your colleagues.

To access ROOT files via a web server, create a TWebFile object instead of a TFile object with a standard URL as file name. For example:

```
root [] TWebFile f("http://root.cern.ch/~rdm/hsimple.root")
root [] 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 px versus px
KEY: TNtuple ntuple;1 Demo ntuple
root [] 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 read-only file. A TWebFile object is ideally suited to read relatively small objects (like histograms or other data analysis results). Although possible, you don't want to analyze large TTree's via a TWebFile.

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

1. Go to your Apache source directory and add the file
   \texttt{ftp://root.cern.ch/root/mod_root.c} or \texttt{ftp://root.cern.ch/root/mod_root133.c}
   when your Apache server is > 1.2 (rename the file \texttt{mod_root.c}).
2. Add to the end of the Configuration file the line:
   \texttt{Module root_module mod_root.o}
3. Run the Configure script
4. Type make
5. Copy the new httpd to its expected place
6. Go to the conf directory and add at the end of the \texttt{srm.conf} file the line:
   \texttt{AddHandler root-action root}
7. Restart the httpd server

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:

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

Depending on the name argument, the function returns 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.
12 Trees

Why should you Use a Tree?

In the Input/Output chapter, we saw how objects can be saved in ROOT files. In case you want to store large quantities of same-class objects, ROOT has designed the TTree and TNtuple classes specifically for that purpose. The TTree class is optimized to reduce disk space and enhance access speed. A TNtuple is a TTree that is limited to only hold floating-point numbers; a TTree on the other hand can hold all kind of data, such as objects or arrays in addition to all the simple types.

When using a TTree, we fill its branch buffers with leaf data and the buffers are written to file when it is full. Branches, buffers, and leafs, are explained a little later in this chapter, but for now, it is important to realize that not each object is written individually, but rather collected and written a bunch at a time. This is where the TTree takes advantage of compression and will produce a much smaller file than if the objects were written individually. Since the unit to be compressed is a buffer, and the TTree contains many same-class objects, the header of the objects can be compressed. The TTree reduces the header of each object, but it still contains the class name. Using compression, the class name of each same-class object has a good chance of being compressed, since the compression algorithm recognizes the bit pattern representing the class name. Using a TTree and compression the header is reduced to about 4 bytes compared to the original 60 bytes. However, if compression is turned off, you will not see these large savings.

The TTree is also used to optimize the data access. A tree uses a hierarchy of branches, and each branch can be read independently from any other branch. Now, assume that $P_x$ and $P_y$ are data members of the event, and we would like to compute $P_x^2 + P_y^2$ for every event and histogram the result. If we had saved the million events without a TTree we would have to: 1) read each event in its entirety into memory, 2) extract the $P_x$ and $P_y$ from the event, 3) compute the sum of the squares, and 4) fill a histogram. We would have to do that a million times! This is very time consuming, and we really do not need to read the entire event, every time. All we need are two little data members ($P_x$ and $P_y$). On the other hand, if we use a tree with one branch containing $P_x$ and another branch containing $P_y$, we can read all values of $P_x$ and $P_y$ by only reading the $P_x$ and $P_y$ branches. This makes the use of the TTree very attractive.

A Simple TTree

This script builds a TTree from an ASCII file containing statistics about the staff at CERN. This script, staff.C and its input file staff.dat are in $ROOTSYS/tutorials.

```c
// example of macro to read data from an ascii file and
// create a root file with an histogram and a TTree.
gROOT->Reset();

// the structure to hold the variables for the branch
struct staff_t {
  Int_t cat;
  Int_t division;
  Int_t flag;
  Int_t age;
  Int_t service;
  Int_t children;
  Int_t grade;
  Int_t step;
  Int_t nation;
  Int_t hrweek;
  Int_t cost;
};
staff_t staff;

// open the ASCII file
FILE *fp = fopen("staff.dat","r");
char line[81];
// create a new ROOT file
TFile *f = new TFile("staff.root","RECREATE");
// create a TTree
TTree *tree = new TTree("tree",
  "staff data from ascii file");
// create one branch with all the information from
// the structure
tree->Branch("staff",&staff.cat,"cat/I:division:
  flag:age:service:children:grade:step:
  nation:hrweek:cost");
// fill the tree from the values in ASCII file
while (fgets(line,80,fp)) {
  sscanf(&line[0], "%d%d%d%d", &staff.cat,&staff.division,&staff.flag,&staff.age);
  sscanf(&line[13], "%d%d", &staff.service,&staff.children);
  sscanf(&line[24], "%d", &staff.grade);
  sscanf(&line[24], "%d", &staff.step);
  sscanf(&line[24], "%d", &staff.nation);
  sscanf(&line[24], "%d", &staff.hrweek);
  sscanf(&line[24], "%d", &staff.cost);
  tree->Fill();
}
// check what the tree looks like
tree->Print();
fclose(fp);
f->Write();
```
The script declares a variable named staff_t, with several integers representing the relevant attribute of a staff member.

The script opens an ASCII file, creates a ROOT file and a TTree. Then it creates one branch with the TTree::Branch method.

The first parameter of the Branch method is the branch name. The second parameter is the address from which the first leaf is to be read. In this example, it is the address of the structure staff.

Once the branch is defined, the script reads the data from the ASCII file into the staff_t structure and fills the tree.

The ASCII file is closed, and the ROOT file is written to disk saving the tree.

Remember, trees and histograms are created in the current directory, which is the file in our example. Hence an f->Write() saves the tree.

Show an Entry with TTree::Show

An easy way to access one entry of a tree is the use of the TTree::Show method. For example, to look at the 10th entry in the staff.root tree:

```
root [] TFile f("staff.root")
root [] tree->Show(10)
```  
```
cat = 361
division = 9
flag = 15
age = 51
service = 29
children = 0
grade = 7
step = 13
nation = 7
hrweek = 40
cost = 7599
```

Print the tree structure with TTree::Print

A helpful command to see the tree structure meaning the number of entries, the branches and the leaves, is TTree::Print:

```
root [] tree->Print()
```  
```
Entries: 3354 : Total = 344880 bytes  File Size = 46302
 Tree compression factor = 3.24

* 0 : staff
   cat : I
   division : I
   flag : I
   age : I
   service : I
   children : I
   grade : I
   step : I
   nation : I
   hrweek : I
   cost : I

* 5 : 9
```

Scan a Variable the Tree with TTree::Scan

The TTree::Scan method shows all values of the list of leaves separated by a colon.

```
root (11) tree->Scan("cost:age:children")

Row | cost | age | children |
--- |-----|-----|---------|
   | 11975 | 58 | 0 |
   | 10228 | 63  | 0 |
   | 10730 | 56 | 2 |
   | 9311 | 61 | 0 |
   | 9666 | 52 | 2 |
   | 7599 | 60 | 0 |
   | 9868 | 53 | 1 |
   | 8012 | 60 | 1 |
```

The Tree Viewer

The tree viewer, a quick and easy way to examine a tree.

To start the tree viewer, open a file and object browser.

Right click on a TTree and select StartViewer.

You can also start the tree viewer from the command line. First load the viewer library:

```
root [] gSystem->Load("libTreePlayer.so")
root [] new TTreeViewer()
```
Here is what the tree viewer looks like for the example file `staff.root`.

The left panel contains the list of trees and their branches, in this case there is only one tree. You can add more trees with the File-Open command to open the file containing the new tree, then use the context menu on the right panel, select `SetTreeName` and enter the name of the tree to add.

On the right are the leaves or variables in the tree. You can double click on any leaf to add it.

To add more than one dimension you can drag and drop any leaf to the X, Y, and Z "boxes". Then push the Draw button, which is marked with the purple icon on the bottom left.

To add a cut/weight to the histogram, enter an expression in the "cut box". The cut box is the one with the scissor icon.

You can create a new expression by right clicking on any of the E() boxes. The expression can be dragged and dropped into any of the boxes (X, Y, Z, Cut, or Scan).

To scan one or more variables, drop them into the Scan box, then double click on the box. You can also redirect the result of the scan to a file by checking the Scan box on top.

When the "Rec" box is checked, the Draw and Scan commands are recorded in the history file and echoed on the command line.

The "Histogram" text box contains the name of the resulting histogram. By default it is `htemp`. You can type any name, if the histogram does not exist it will create one.

The Option text box contains the list of Draw options (see Draw Options in the Histogram Chapter). You can select the options with the Options menu.

The Command box lets you enter any command that you could also enter on the command line.

The vertical slider on the far left side can be used to select the minimum and maximum of an event range. The actual start and end index are shown in the bottom in the status window.

The `IList` and `OList` are to specify an input list of entry indices and a name for the output list respectively. Both must be of type `TList` and contain integers of entry indices. These lists are described below in the paragraph "Creating an Event List".

There is an extensive help utility accessible with the Help menu.

Here are a couple of graphs. The first is a plot of the age distribution, the second a scatter plot of the cost vs. age. The second one was generated by dragging the age leaf into the Y-box and the cost leaf into the X-box, and pressing the Draw button. By default this will generate a scatter plot. Select a different option, for example "lego" to create a 2D histogram.
Creating and Saving Trees

This picture shows the TTree class:

```
Creating a Tree from a Folder Hierarchy
```

An alternative way to create a tree and organize it, is to use folders. You can build a folder structure (see the chapter on Folders and Tasks), and create a tree with branches for each of the sub-folders:

```
TTree folder_tree("MyFolderTree", "/MyFolder")
```

The second argument is the top folder, and the "/" signals the TTree constructor that this is a folder not just the title. You fill the tree by placing the data into the folder structure and calling TTree::Fill.

The reverse is also true, one can recreate the folder hierarchy from the tree with the TTree::SetFolder method.

Autosave

Autosave gives the option to save all branch buffers every n byte. We recommend using Autosave for large acquisitions. If the acquisition fails to complete, you can recover the file and all the contents since the last Autosave. To set the number of bytes between Autosave you can use the TTree::SetAutosave() method. You can also call TTree::Autosave in the acquisition loop every n entry.

Branches

The class for a branch is called TBranch. The organization of branches allows the designer to optimize the data for the anticipated use.

If two variables are independent, and the designer knows the variables will not be used together, she would place them on separate branches. If, however, the variables are related, such as the coordinates of a point, it is most efficient to create one branch with both coordinates on it. A variable on a TBranch is called a leaf (yes - TLeaf).

Another point to keep in mind when designing trees is the branches of the same TTree can be written to separate files.

To add a TBranch to a TTree we call the TTree::Branch() method. Note that we DO NOT use the TBranch constructor.

The TTree::Branch method has several signatures. The branch type differs by what is stored in it. A branch can hold an entire object, a list of simple variables, contents of a folder, contents of a TList, or an array of objects. Let's see some examples.

To follow along you will need the shared library libEvent.so. First, check if it is in $ROOTSYS/test. If it is, copy it to your own area. If it is not there, you have to build it.
Adding a Branch to hold a List of Variables

As in the very first example (staff.root) the data we want to save is a list of simple variables, such as integers or floats. In this case, we use the following TBranch signature:

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

The first parameter is the branch name.
The second parameter is the address from which the first variable is to be read. In the code above, "event" is a structure with one float and three integers and one unsigned integer.

You should not assume that the compiler aligns the elements of a structure without gaps. To avoid alignment problems, you need to use structures with same length members. If your structure does not qualify, you need to create one branch for each element of the structure.

The leaf name is NOT used to pick the variable out of the structure, but is only used the name for the leaf. This means that the list of variables needs to be in a structure in the order described in the third parameter.

This third parameter is a string describing the leaf list. Each leaf has a name and a type separated by a "/" and it is separated from the next leaf by a ":".

```
<Variable>/<type>:<Variable>/<type>
```

The example on the next line has two leaves: a floating-point number called temp and an integer named ntrack.

```
"temp/F:ntrack/I":
```

The type can be omitted and if no type is given, the same type as the previous variable is assumed. This leaf list has three integers called ntrack, nseg, and nvtex.

```
*ntrack/I:nseg:nvtex*
```

There is one more rule: when no type is given for the very first leaf, it becomes a float (F). This leaf list has three floats called temp, mass, and px.

```
"temp:mass:px"
```

The symbols used for the type are:

- C: a character string terminated by the 0 character.
- B: an 8 bit signed integer.
- b: an 8 bit unsigned integer.
- S: a 16 bit signed integer.
- s: a 16 bit unsigned integer.
- I: a 32 bit signed integer.
- i: a 32 bit unsigned integer.
- F: a 32 bit floating point.
- D: a 64 bit floating point.

Adding a TBranch to hold an Object

To write a branch to hold an event object, we need to load the definition of the Event class, which is in $ROOTSYS/test/libEvent.so.

For an object to be in a tree it's class definition needs to include the ClassDef/ClassImp macros. We expect to remove this restriction in the near future.

```
root [] .L libEvent.so
```

First, we need to open a file and create a tree.

```
root [] TFile *f = new TFile("AFile.root", "RECREATE");
root [] TTree *tree = new TTree("T","A Root Tree")
```

The type is used for a byte count to decide how much space to allocate. The variable written is simply the block of bytes starting at the starting address given in the second parameter. It may or may not match the leaf list depending on whether or not the programmer is being careful when choosing the leaf address, name, and type.

By default, a variable will be copied with the number of bytes specified in the type descriptor symbol. However, if the type consists of two characters, the number specifies the number of bytes to be used when copying the variable to the output buffer. The line below describes ntrack to be written as a 16-bit integer (rather than a 32-bit integer).

```
*ntrack/I2*
```

With this Branch method, you can also add a leaf that holds an entire array of variables. To add an array of floats use the f[n] notation when describing the leaf.

```
Float_t *f[10];
tree->Branch("fBranch", &f[10]/F);
```

You can also add an array of variable length:

```
{
    TFile *f = new TFile("peter.root", "recreate");
    Int_t nPhot;
    Float_t *f;
    TTree* nEmcPhotons = new TTree("nEmcPhotons","EMC Photons");
    nEmcPhotons->Branch("nPhot", &nPhot,
    E[nPhot]/F);
}
```

For an example see Example 2 below ($ROOTSYS/tutorials/tree2.C) and staff.C at the beginning of this chapter.
We need to create a pointer to an Event object that will be used as a reference in the TTree::Branch method. Then we create a branch with the TTree::Branch method.

```c++
root[0] Event *event = new Event();
root[0] tree->Branch("EventBranch","Event", &event, 32000, 99)
```

To add a branch to hold an object we use the signature above. The first parameter is the name of the branch. The second parameter is the name of the class of the object to be stored. The third parameter is the address of a pointer to the object to be stored.

Note that it is an address of a pointer to the object, not just a pointer to the object.

The fourth parameter is the buffer size and is by default 32000 bytes. It is the number of bytes of data for that branch to save to a buffer until it is saved to the file.

The last parameter is the split-level, which is the topic of the next section.

Static class members are not part of an object and thus not written with the object. You could store them separately by collecting these values in a special “status” object and write it to the file outside of the tree. If it makes sense to store them for each object, make them a regular data member.

**Setting the Split-level**

To split a branch means to create a sub-branch for each data member in the object. The split-level can be set to 0 to disable splitting or it can be set to a number between 1 and 99 indicating the depth of splitting.

If the split-level is set to zero, the whole object is written in its entirety to one branch. The TTree will look like the one on the right, with one branch and one leaf holding the entire event object.

A tree that is split  
A tree that is not split

When the split level is 1, an object data member is assigned a branch. If the split level is 2, the data member objects will be split also, and a split level of 3 its data members objects, will be split. As the split level increases so does the splitting depth.

ROOT’s default for the split level is 99, this means the object will be split to the maximum.

**Memory Considerations when Splitting a Branch**

Splitting a branch can quickly generate many branches. Each branch has its own buffer in memory. In case of many branches (say more than 100), you should adjust the buffer size accordingly. A recommended buffer size is 32000 bytes if you have less than 50 branches. Around 16000 bytes if you have less than 100 branches and 4000 bytes if you have more than 500 branches. These numbers are recommended for computers with memory size ranging from 32MB to 256MB. If you have more memory, you should specify larger buffer sizes. However, in this case, do not forget that your file might be used on another machine with a smaller memory configuration.

**Performance Considerations when Splitting a Branch**

A split branch is faster to read, but slightly slower to write. The reading is quicker because variables of the same type are stored consecutively and the type does not have to be read each time. It is slower to write because of the large number of buffers as described above. See Performance Benchmarks for performance impact of split and non-split mode.

**Rules for Splitting**

When splitting a branch, variables of different types are handled differently. Here are the rules that apply when splitting a branch.

- If a data member is a basic type, it becomes one branch of class TBranchElement.
- A data member can be an array of basic types. In this case, one single branch is created for the array.
- A data member can be a pointer to an array of basic types. The length can vary, and must be specified in the comment field of the data member in the class definition. (see I/O chapter).
- Pointer data member are not split, except for pointers to a TClonesArray. The TClonesArray (pointed to) is split if the split level is greater than two. When the split level is one, the TClonesArray is not split.
- If a data member is a pointer to an object, a special branch is created. The branch will be filled by calling the class Streamer function to serialize the object into the branch buffer.
- If a data member is an object, the data members of this object are split into branches according to the split level (i.e. split level > 2).
- Base classes are split when the object is split.
- Abstract base classes are never split
- Most STL containers are supported except for some extreme cases. These examples are not supported:

```c++
// STL vector of vectors of TAxis*
vector<vector<TAxis *> > fVectAxis;
// STL map of string/vector
map<string,vector<int> > fMapString;
// STL deque of pair
deque<pair<float,float> > fDequePair;
```

- C-structure data members are not supported in split mode.
- An object that is not split may be slow to browse.
- An STL container that is not split will not be accessible in the browser.
Exempt a Data Member from Splitting

If you are creating a branch with an object and in general you want the data members to be split, but you want to exempt a data member from the split. You can specify this in the comment field of the data member:

```cpp
class Event : public TObject {
    private:
    EventHeader fEvtHdr;    ///// Don't split the header
}
```

Adding a Branch to hold a TClonesArray

ROOT has two classes to manage arrays of objects. The TObjectArray that can manage objects of different classes, and the TClonesArray that specializes in managing objects of the same class (hence the name Clones Array). TClonesArray takes advantage of the constant size of each element when adding the elements to the array. Instead of allocating memory for each new object as it is added, it reuses the memory. Here is an example of the time a TClonesArray can save over a TObjectArray.

We have 100,000 events, and each has 10,000 tracks, which gives 1,000,000,000 tracks. If we use a TObjectArray for the tracks, we implicitly make a call to new and a corresponding call to delete for each track. The time it takes to make a pair of new/delete calls is about 7 μs (10^-6). If we multiply the number of tracks by 7 μs, (1,000,000,000 * 7 * 10^-6) we calculate that the time allocating and freeing memory is about 2 hours. This is the chunk of time saved when a TClonesArray is used rather than a TObjectArray. If you don't want to wait 2 hours for your tracks (or equivalent objects), be sure to use a TClonesArray for same-class objects arrays.

Branches with TClonesArrays use the same method (TTree::Branch) as any other object described above. If splitting is specified the objects in the TClonesArray are split, not the TClonesArray itself.

Identical Branch Names

When a top-level object (say event), has two data members of the same class the sub branches end up with identical names. To distinguish the sub branch we must associate them with the master branch by including a "\" (doll) at the end of the master branch name. This will force the name of the sub branch to be master.sub branch instead of simply sub branch.

For example, a tree has two branches Trigger and MuonTrigger, each containing an object of the same class (Trigger). To uniquely identify the sub branches we add the dot:

```cpp
tree->Branch("Trigger.",&b1,8000,1);
tree->Branch("MuonTrigger.",&b2,8000,1);
```

If Trigger has three members, T1, T2, T3, the two instructions above will generate sub branches called:

```cpp
Trigger.T1, Trigger.T2, Trigger.T3, 
MuonTrigger.T1, MuonTrigger.T2, MuonTrigger.T3.
```

Adding a Branch with a Folder

To add a branch from a folder use the syntax:

```cpp
tree->Branch("/aFolder");
```

This method creates one branch for each element in the folder. The method returns the total number of branches created.

Adding a Branch with a Collection

This Branch method creates one branch for each element in the collection.

```cpp
tree->Branch("aCollection, B000, 99");
// Int_t TTree::Branch(TCollection *list, Int_t bufsize, // Int_t splitlevel, const char *name)
```

The method returns the total number of branches created. Each entry in the collection becomes a top level branch if the corresponding class is not a collection. If it is a collection, the entry in the collection becomes in turn top level branches, etc. The split level is decreased by 1 every time a new collection is found. For example if list is a TObjectArray'

- if splitlevel = 1, one top level branch is created for each element of the TObjectArray.
- if splitlevel = 2, one top level branch is created for each array element. If, in turn, one of the array elements is a TCollection, one top level branch will be created for each element of this collection.

In case a collection element is a TClonesArray, the special Tree constructor for TClonesArray is called. The collection itself cannot be a TClonesArray.

If name is given, all branch names will be prefixed with name_.

IMPORTANT NOTE1: This function should not be called with splitlevel < 1.

IMPORTANT NOTE2: The branches created by this function will have names corresponding to the collection or object names. It is important to give names to collections to avoid misleading branch names or identical branch names. By default collections have a name equal to the corresponding class name, e.g. the default name for a TList is "List".

Examples For Writing and Reading Trees

The following sections are examples of writing and reading trees increasing in complexity from a simple tree with a few variables to a tree containing folders and complex Event objects.

Each example has a named script in the $ROOTSYS/tutorials directory. They are called tree1.C to tree4.C. The examples are:

- tree1.C: A tree with several simple (integers and floating point) variables.
- tree2.C: A tree built from a C structure (struct). This example uses the Geant3 C wrapper as an example of a Fortran common block ported to C with a C structure.
• tree3.C: In this example we will show how to extend a tree with a branch from another tree with the Friends feature. These trees have branches with variable length arrays. Each entry has a variable number of tracks, and each track has several variables.

• tree4.C: A tree with a class (Event). The class Event is defined in $ROOTSYS/test. In this example we first encounter the impact of splitting a branch.

Each script contains the main function, with the same name as the file (i.e. tree1), the function to write - tree1w, and the function to read - tree1r. If the script is not run in batch mode, it displays the tree in the browser and tree viewer.

To study the example scripts, you can either execute the main script, or load the script and execute a specific function. For example:

```c
void tree1w()
{
//create a Tree file tree1.root
//create the file, the Tree and a few branches
TFile f("tree1.root","recreate");
TTree t1("t1","a simple Tree with simple variables");
Float_t px, py, pz;
Double_t random;
Int_t ev;
t1.Branch("px",&px,"px/F");
t1.Branch("py",&py,"py/F");
t1.Branch("pz",&pz,"pz/F");
t1.Branch("ev",&ev,"ev/I");
// continued on the next page ...
```

### Example 1: A Tree with Simple Variables

This example shows how to write, view, and read a tree with several simple (integers and floating point) variables.

#### Writing the Tree

Below is the function that writes the tree (tree1w). First, the variables are defined (px, py, pz, random and ev). Then we add a branch for each of the variables to the tree, by calling the `TTree::Branch` method for each variable.

```c
void tree1w()
{
//create a Tree file tree1.root
//create the file, the Tree and a few branches
TFile f("tree1.root","recreate");
TTree t1("t1","a simple Tree with simple variables");
Float_t px, py, pz;
Double_t random;
Int_t ev;
t1.Branch("px",&px,"px/F");
t1.Branch("py",&py,"py/F");
t1.Branch("pz",&pz,"pz/F");
```

```c
// continued on the next page ...
```

#### Filling the Tree

First we find some random values for the variables. We assign px and py a gaussian with mean = 0 and sigma = 1 by calling `gRandom->Rannor(px,py)`, and calculate pz. Then we call the `TTree::Fill` method. Because we have already organized the tree into branches and told each branch where to get the value from, the call `t1.Fill()`, fills all branches in the tree.

After this script is executed we have a ROOT file called `tree1.root` with a tree called `t1`. 

```c
for (Int_t i=0;i<10000;i++) {
    gRandom->Rannor(px,py);
    pz = px*px + py*py;
    random = gRandom->Rndm();
    ev = i;
    t1.Fill();
}
```

// continued from previous page

//fill the tree
for (Int_t i=0;i<10000;i++) {
    gRandom->Rannor(px,py);
    pz = px*px + py*py;
    random = gRandom->Rndm();
    ev = i;
    t1.Fill();
}
```

```
// save the Tree header.  
// The file will be automatically closed 
// when going out of the function scope

t1.Write();
```

### Creating Branches with A single Variable

This is the signature of `TTree::Branch` to create a branch with a list of variables:

```c
TBranch* TTree::Branch(const char* name, void* address, 
const char* leaflist, Int_t bufsize = 32000)
```

The first parameter is the branch name.

The second parameter is the address from which to read the value.

The third parameter is the leaf list with the name and type of each leaf.

In this example each branch has only one leaf. In the box below, the branch is named `px` and has one floating point type leaf also called `px`.

```c
t1.Branch("px",&px,"px/F");
```

```c
// execute the tree1() function 
// that writes, reads, and shows the tree
root[ ].x tree1.C 
// use ACLiC to build a shared library and 
// check syntax, then execute as above
root[ ].x tree1.C++
// load the script and select a function to execute
root[ ].L tree1.C
root[ ] tree1w()
root[ ] tree1r()
```
Viewing the Tree

This is the tree1.root file and its tree in the browser.

In the right panel are the branches ev, px, py, pz, and random. Note that these are shown as leaves because they are "end" branches with only one leaf.

To histogram a leaf we can simply double click on it in the browser:

This is how the tree t1 looks in the Tree Viewer. Here we can add a cut and add other operations for histogramming the leaves (see the section on Tree Viewer). For example, we can plot a two dimensional histogram.

Reading the Tree

The tree1r function shows how to read the tree and access each entry and each leaf.

We first define the variables to hold the read values.

```c
Float_t px, py, pz;
```

Then we tell the tree to populate these variables when reading an entry. We do this with the TTree::SetBranchAddress method. The first parameter is the branch name, and the second is the address of the variable where the branch data is to be placed.

In this example the branch name is px. This name was given when the tree was written (see tree1w). The second parameter is the address of the variable px.

```c
t1->SetBranchAddress("px",&px);
```

GetEntry

Once the branches have been given the address, a specific entry can be read into the variables with the method TTree::GetEntry(n).

The TTree::GetEntry method reads all the branches for entry (n) and populates the given address accordingly.

By default, GetEntry() reuses the space allocated by the previous object for each branch. You can force the previous object to be automatically deleted if you call mybranch.SetAutoDelete(kTRUE) (default is kFALSE).

Example:
Consider the example in $ROOTSYS/test/Event.h. The top level branch in the tree t is declared with:

```c
Event *event = 0;
//event must be null or point to a valid object
//it must be initialized
T.SetBranchAddress("event",&event);
```

When reading the Tree, one can choose one of these 3 options:

Option 1:

```c
for (Int_t i=0;i<nentries;i++) {
  T.GetEntry(i);
  //the object event has been filled at this point
  Event *event = 0;
  //event must be null or point to a valid object
  //it must be initialized
  T.SetBranchAddress("event",event);
}
```

The is the default and recommended. At the first entry an object of the class Event will be created and pointed by event.

At the following entries, event will be overwritten by the new data. All internal members that are TObject* are automatically deleted. It is important
that these members be in a valid state when GetEntry is called. Pointers
must be correctly initialized.

However these internal members will not be deleted if the characters ’->’
are specified as the first characters in the comment field of the data member
declaration.

If ’->’ is specified, the pointer member is read via pointer-
>Streamer(buf). In this case, it is assumed that the pointer is never null
(see pointer TClonesArray *fTracks in the $ROOTSYS/test/Event
example).

If ’->’ is not specified, the pointer member is read via buf >> pointer. In
this case the pointer may be null. Note that the option with ’->’ is faster to
read or write and it also consumes less space in the file.

Option 2:
The option AutoDelete is set:

TBranch *branch = T.GetBranch("event");
branch->SetAddress(&event);
for (Int_t i=0;i<nentries;i++) {
  T.GetEntry(i);
  // the object event has been filled at this point
}

In this case, at each iteration, the object event is deleted by GetEntry and
a new instance of Event is created and filled.

Option 3:
Same as option 1, but you delete yourself the event:

for (Int_t i=0;i<nentries;i++) {
  delete event;
  event = 0;
  T.GetEntry(i);
  // the object event has been filled at this point
}

It is strongly recommended to use the default option 1. It has the additional
advantage that functions like TTree::Draw (internally calling
TTree::GetEntry) will be functional even when the classes in the file are
not available.

Reading selected branches is quicker than reading an entire entry. If you are
interested in only one branch, you can use the TBranch::GetEntry
method and only that branch is read.

Here is the script tree1r:

```c
void tree1r()
{
  //read the Tree generated by tree1w
  //and fill two histograms

  //note that we use "new" to create the TFile
  //and TBranch objects, because we want to keep
  //these objects alive when we leave this function.
  TFile *f = new TFile("tree1.root");
  TTree *t1 = (TTree*)f->Get("t1");
  Float_t px, py, pz;
  Double_t random;
  Int_t ev;
  t1->SetBranchAddress("px", &px);
  t1->SetBranchAddress("py", &py);
  t1->SetBranchAddress("pz", &pz);
  t1->SetBranchAddress("random", &random);
  t1->SetBranchAddress("ev", &ev);

  //create two histograms
  TH1F *hpx = new TH1F("hpx","px distribution",100,-3,3);
  TH2F *hpxpy = new TH2F("hpxpy","py vs px",30,-3,3,30,-3,3);

  //read all entries and fill the histograms
  Int_t nentries = (Int_t)t1->GetEntries();
  for (Int_t i=0;i<nentries;i++) {
    t1->GetEntry(i);
    hpx->Fill(px);
    hpxpy->Fill(px,py);
  }
  //we do not close the file.
  //We want to keep the generated histograms
  //we open a browser and the TreeViewer
  if (gROOT->IsBatch()) return;
  new TBrowser();
  t1->StartViewer();
  //In the browser, click on "ROOT Files",
  //then on "tree1.root".
  //You can click on the histogram icons
  //in the right panel to draw them.
  //In the TreeViewer, follow the instructions
  //in the Help button.
}
```
Example 2: A Tree with a C Structure

The executable script for this example is `$ROOTSYS/tutorials/tree2.C`. In this example we show:

- how to build branches from a C structure
- how to make a branch with a fixed length array
- how to make a branch with a variable length array
- how to read selective branches
- how to fill a histogram from a branch
- how to use `TTree::Draw` to show a 3D plot.

A C structure (`struct`) is used to build a ROOT tree. In general we discourage the use of C structures, we recommend using a class instead. However, we do support them for legacy applications written in C or Fortran.

The example `struct` holds simple variables and arrays. It maps to a Geant3 common block `/gctrak/`. This is the definition of the common block/structure:

```c
const Int_t MAXMEC = 30;
// PARAMETER (MAXMEC=30)
// COMMON/GCTRAK/VECT(7), GETOT, GEKIN, VOUT(7)
// + , NMMEC (MAXMEC)
// + , LMEC (MAXMEC), NSTEP
// + , PID, DSTEP, DESTEL, SAFETY, SLENG
// + , STEP, SNEXT, SFIELD, TOFG, GEKRAT, UPWGT

typedef struct {
  Float_t  vect[7];
  Float_t  getot;
  Float_t  gekin;
  Float_t  vout[7];
  Int_t    nmec;
  Int_t    lme[MAKMAXMEC];
  Int_t    namec[MAKMAXMEC];
  Int_t    nstep;
  Int_t    pid;
  Float_t  dstep;
  Float_t  destel;
  Float_t  safety;
  Float_t  sleng;
  Float_t  step;
  Float_t  snext;
  Float_t  sfeld;
  Float_t  tofg;
  Float_t  gekrat;
  Float_t  upwgt;
} Gctrak_t;
```

When using Geant3, the common block is filled by Geant3 routines at each step and only the `Tree::Fill` method needs to be called. In this example we emulate the Geant3 step routine with the `helixStep` function. We also emulate the filling of the particle values. The calls to the `Branch` methods are the same as if Geant3 were used.

```c
void helixStep(Float_t step, Float_t *vect, Float_t *vout)
{
  // extrapolate track in constant field
  Float_t field = 20; // field in kilogauss
  enum Evect {kX,kY,kZ,kPX,kPY,kPZ,kPP};
  vout[kPP] = vect[kPP];
  Float_t h4 = field*2.99792e-4;
  Float_t rho = -h4/vect[kPP];
  Float_t tsint = rho*step;
  Float_t sintt = 1 - tsint;
  Float_t sint = rho*step;
  Float_t cos1t = tsint/2;
  Float_t f1 = step*sintt;
  Float_t f2 = step*cos1t;
  Float_t f3 = step*tsint*vect[kPZ];
  Float_t f4 = -step*cos1t;
  Float_t f5 = sint;
  Float_t f6 = step*cos1t*vect[kPZ];
  vout[kX] = vect[kX] + (f1*vect[kPX] - f2*vect[kPY]);
  vout[kY] = vect[kY] + (f1*vect[kPY] + f2*vect[kPX]);
  vout[kZ] = vect[kZ] + (f1*vect[kPZ] + f3);
  vout[kPX] = vect[kPX] + (f4*vect[kPX] - f5*vect[kPY]);
  vout[kPY] = vect[kPY] + (f4*vect[kPY] + f5*vect[kPX]);
  vout[kPZ] = vect[kPZ] + (f4*vect[kPZ] + f6);
}
```
void tree2w() // write tree2 example
{
    //create a Tree file tree2.root
    TFile f("tree2.root","recreate");
    //create the file, the Tree
    TTree t2("t2","a Tree with data from a fake Geant3");
    // declare a variable of the C structure type
    Gctrak_t gstep;
    // add the branches for a subset of gstep
    t2.Branch("vect",gstep.vect,"vect[7]/F");
    t2.Branch("getot",&gstep.getot,"getot/F");
    t2.Branch("gekin",&gstep.gekin,"gekin/F");
    t2.Branch("nmec",&gstep.nmec,"nmec/F");
    t2.Branch("destep",&gstep.destep,"destep/F");
    t2.Branch("pid",&gstep.pid,"pid/I");
    //Initialize particle parameters at first point
    Float_t px,py,pz,p,charge=0;
    Float_t vout[7];
    Float_t mass  = 0.137;
    Bool_t newParticle = kTRUE;
    gstep.step    = 0.1;
    gstep.destep  = 0;
    gstep.nmec    = 0;
    gstep.pid     = 0;
    //transport particles
    for (Int_t i=0; i<10000; i++) {
        //generate a new particle if necessary
        // (Geant3 emulation)
        if (newParticle) {
            px = gRandom->Gaus(0,.02);
            py = gRandom->Gaus(0,.02);
            pz = TMath::Sqrt(px*px+py*py+pz*pz);
            charge = 1; if (gRandom->Rndm() < 0.5) charge = -1;
            gstep.pid    += 1;
            gstep.vect[0] = 0;
            gstep.vect[1] = 0;
            gstep.vect[2] = 0;
            gstep.vect[3] = px/p;
            gstep.vect[4] = py/p;
            gstep.vect[5] = pz/p;
            gstep.vect[6] = p*charge;
            gstep.getot = TMath::Sqrt(p*p + mass*mass);
            gstep.gekin = gstep.getot - mass;
            newParticle = kFALSE;
        }
        // transported particle in magnetic field
        // (Geant3 emulation)
        helixStep(gstep.step, gstep.vect, vout);
        // make one step
        t2.Fill();
        //apply energy loss
        gstep.destep   = gstep.step*gRandom->Gaus(0.0002,0.00001);
        gstep.gekin -= gstep.destep;
        gstep.getot  = gstep.gekin + mass;
        gstep.vect[6]=charge*TMath::Sqrt(gstep.getot*gstep.getot - mass*mass);
        gstep.vect[0] = vout[0];
        gstep.vect[1] = vout[1];
        gstep.vect[2] = vout[2];
        gstep.vect[3] = vout[3];
        gstep.vect[4] = vout[4];
        gstep.vect[5] = vout[5];
        gstep.nmec    = (Int_t)(5*gRandom->Rndm());
        for (Int_t l=0;l<gstep.nmec;l++) gstep.lmec[l] = l;
        if (gstep.gekin < 0.001) newParticle = kTRUE;
        if (TMath::Abs(gstep.vect[2]) > 30)
            newParticle = kTRUE;
    }
    // save the Tree header. The file will be automatically
    // closed when going out of the function scope
    t2.Write();
}

// fill the Tree with current step parameters
//t2.Fill();

//transport particle in magnetic field
// (Geant3 emulation)
helixStep(gstep.step, gstep.vect, vout); // make one step

//apply energy loss
//gstep.destep = gstep.step*gRandom->Gaus(0.0002,0.00001);
gstep.gekin = gstep.destep;
gstep.getot = gstep.gekin + mass;
gstep.vect[6] = charge*TMath::Sqrt(gstep.getot*gstep.getot - mass*mass);
gstep.vect[0] = vout[0];
gstep.vect[1] = vout[1];
gstep.vect[2] = vout[2];
gstep.vect[3] = vout[3];
gstep.vect[4] = vout[4];
gstep.vect[5] = vout[5];
gstep.nmec = (Int_t)(5*gRandom->Rndm());
for (Int_t l=0;l<gstep.nmec;l++) gstep.lmec[l] = l;
if (gstep.gekin < 0.001) newParticle = kTRUE;
if (TMath::Abs(gstep.vect[2]) > 30)
    newParticle = kTRUE;
}
// closed when going out of the function scope
//t2.Write();

Adding a Branch with a Fixed Length Array
At first, we create a tree and create branches for a subset of variables in the
C structure Gctrak_t. Then we add several types of branches.
The first branch reads seven floating point values beginning at the address of
'gstep.vect'. You do not need to specify &gstep.vect, because in C
and C++ the array variable holds the address of the first element.

t2.Branch("vect",gstep.vect,"vect[7]/F");
t2.Branch("getot",&gstep.getot,"getot/F");
t2.Branch("gekin",&gstep.gekin,"gekin/F");

Adding a Branch with a Variable Length Array
The next two branches are dependent on each other. The first holds the
length of the variable length array and the second holds the variable length
array.
The nmec branch reads nmec number of integers beginning at the address
gstep.destep.

t2.Branch("nmec",&gstep.nmec,"nmec/I");
t2.Branch("lmec",gstep.lmec,"lmec[nmec]/I");
The variable \texttt{nmec} is a random number and is reset for each entry.

\begin{verbatim}
gstep.nmec = (Int_t)(5*GRandom->Rndm());
\end{verbatim}

\textbf{Filling the Tree}

In this emulation of Geant3, we generate and transport particles in a magnetic field and store the particle parameters at each tracking step in a ROOT tree.

\textbf{Analysis}

In this analysis we do not read the entire entry, we only read one branch. First we set the address for the branch to the file \texttt{dstep}, then we use the \texttt{TBranch::GetEntry} method.

Then we fill a histogram with the \texttt{dstep} branch entries, draw it and fit it with a gaussian.

In addition we draw the particle's path using the three values in the vector. Here we use the \texttt{TTree::Draw} method. It automatically creates a histogram and plots the 3 expressions (see Using Trees in Analysis).

\begin{verbatim}
void tree2r()
{
    // read the Tree generated by tree2w and fill one histogram
    // we are only interested by the destep branch.
    // note that we use "new" to create the TFile and TTree objects
    // because we want to keep these objects alive when we leave
    // this function.
    TFile *f = new TFile("tree2.root");
    TTree *t2 = (TTree*)f->Get("t2");
    static Float_t destep;
    TBranch *b_destep = t2->GetBranch("destep");
    b_destep->SetAddress(&destep);
    //create one histogram
    TH1F *hdestep = new TH1F("hdestep","destep in Mev",100,1e-5,3e-5);
    //read only the destep branch for all entries
    Int_t nentries = (Int_t)t2->GetEntries();
    for (Int_t i=0;i<nentries;i++) {
        b_destep->GetEntry(i);
        // fill the histogram with the destep entry
        hdestep->Fill(destep);
    }
    // we do not close the file.
    // We want to keep the generated histograms
    // We fill a 3-d scatter plot with the particle
    // step coordinates
    TCanvas *c1 = new TCanvas("c1","c1",600,800);
    c1->SetFillStyle(45);
    hdestep->Fit("gaus");
    c1->cd(2);
    gPad->SetFillStyle(37);
    gPad->SetLineWidth(4);
    t2->Draw("vect[0]:vect[1]:vect[2] ");
    if (gROOT->IsBatch()) return;
    // invoke the x3d viewer
    gPad->x3d();
}
\end{verbatim}
Example 3: Adding Friends to Trees

In this example we will show how to extend a tree with a branch from another tree with the Friends feature.

Adding a Branch to an Existing Tree

You may want to add a branch to an existing tree. For example, if one variable in the tree was computed with a certain algorithm, you may want to try another algorithm and compare the results.

One solution is to add a new branch, fill it, and save the tree. The code below adds a simple branch to an existing tree.

Note the kOverwrite option in the Write method, it overwrites the existing tree. If it is not specified, two copies of the tree headers are saved.

```
void tree3AddBranch()
{
    TFile f("tree3.root","update");
    Float_t new_v;
    TTree *t3 = (TTree*)f->Get("t3");
    TBranch *newBranch = t3->Branch("new_v", &new_v, "new_v/F");
    Int_t nentries = (Int_t)t3->GetEntries();
    for (Int_t i = 0; i < nentries; i++)
    {
        new_v = gRandom->Gaus(0,1);
        newBranch->Fill();
    }
    t3->Write("", TObject::kOverwrite);
}
```

Adding a branch is often not possible because the tree is in a read-only file and you do not have permission to save the modified tree with the new branch. Even if you do have the permission, you risk loosing the original tree with an unsuccessful attempt to save the modification. Since trees are usually large, adding a branch could extend it over the 2GB limit. In this case, the attempt to write the tree fails, and the original data may also be corrupted.

In addition, adding a branch to a tree enlarges the tree and increases the amount of memory needed to read an entry, and therefore decreases the performance.

For these reasons, ROOT offers the concept of friends for trees (and chains). We encourage you to use TTree::AddFriend rather than adding a branch manually.

**TTree::AddFriend**

A tree keeps a list of friends. In the context of a tree (or a chain), friendship means unrestricted access to the friends data. In this way it is much like adding another branch to the tree without taking the risk of damaging it. To add a friend to the list, you can use the TTree::AddFriend method.
To retrieve the list of friends from a tree use `TTree::GetListOfFriends`.

When the tree is written to file (`TTree::Write`), the friends list is saved with it. And when the tree is retrieved, the trees on the friends list are also retrieved and the friendship restored.

When a tree is deleted, the elements of the friend list are also deleted.

It is possible to declare a friend tree that has the same internal structure (same branches and leaves) as the original tree, and compare the same values by specifying the tree.

```c
TTree.Draw("var1;var2;var3")
```

The example code is in `$ROOTSYS/tutorials/tree3.C`. Here is the script:

```c
void tree3w() {
  // Example of a Tree where branches are variable length arrays
  // A second Tree is created and filled in parallel.
  // Run this script with
  //   .x tree3.C
  // In the function treeer, the first Tree is open.
  // The second Tree is declared friend of the first tree.
  // TTree::Draw is called with variables from both Trees.
  const Int_t kMaxTrack = 500;
  Int_t ntrack;
  Int_t stat[kMaxTrack];
  Int_t sign[kMaxTrack];
  Float_t px[kMaxTrack];
  Float_t py[kMaxTrack];
  Float_t pz[kMaxTrack];
  Float_t zv[kMaxTrack];
  Float_t chi2[kMaxTrack];
  Double_t sumstat;
  // create the first root file with a tree
  TFile f("tree3.root","recreate");
  TTree *t3 = new TTree("t3","Reconst ntuple");
  t3->Branch("ntrack",&ntrack,"ntrack/I");
  t3->Branch("stat",stat,"stat[ntrack]/I");
  t3->Branch("sign",sign,"sign[ntrack]/I");
  t3->Branch("px",px,"px[ntrack]/F");
  t3->Branch("py",py,"py[ntrack]/F");
  t3->Branch("pz",pz,"pz[ntrack]/F");
  t3->Branch("zv",zv,"zv[ntrack]/F");
  t3->Branch("chi2",chi2,"chi2[ntrack]/F");
  // create the second root file with a different tree
  TFile fr("tree3f.root","recreate");
  TTree *t3f = new TTree("t3f","a friend Tree");
  t3f->Branch("ntrack",&ntrack,"ntrack/I");
  t3f->Branch("sumstat",&sumstat,"sumstat/D");
  // Fill the trees
  for (Int_t i=0;i<10000;i++) {
    Int_t nt = gRandom->Rndm()*(kMaxTrack-1);
    ntrack = nt;
    sumstat = 0;
    // set the values in each track
    for (Int_t n=0;n<nt;n++) {
      stat[n] = n3;
      sign[n] = f2;
      px[n] = gRandom->Gaus(0,1);
      py[n] = gRandom->Gaus(0,2);
      pz[n] = gRandom->Gaus(10,5);
      zv[n] = gRandom->Gaus(100,2);
      chi2[n] = gRandom->Gaus(0,0.1);
      sumstat += chi2[n];
      pt[n] = TMath::Sqrt(px[n]*px[n] + py[n]*py[n]);
    }
    t3->Fill();
    t3f->Fill();
  }
  // Write the two files
  t3->Print();
  f.cd();
  t3->Write();
  fr.cd();
  t3f->Write();
}

// Function to read the two files and add the friend
void tree3r() {
  TFile *f = new TFile("tree3.root");
  TTree *t3 = (TTree*)f->Get("t3");
  // Add the second tree to the first tree as a friend
  t3->AddFriend("t3f","tree3f.root");
  // Draw pz which is in the first tree and use pt
  // in the condition. pt is in the friend tree.
  t3->Draw("pz","pt>3");
}

// This is executed when typing .x tree3.C
void tree3() {
  tree3w();
  tree3r();
}
```
Example 4: A Tree with an Event Class

This example is a simplified version of $ROOTSYS/test/MainEvent.cxx and where Event objects are saved in a tree. The full definition of Event is in $ROOTSYS/test/MainEvent.cxx. To execute this macro, you will need the library $ROOTSYS/test/libEvent.so. If it does not exist you can build the test directory applications by following the instruction in the $ROOTSYS/test/README file.

In this example we will show

- the difference in splitting or not splitting a branch
- how to read selected branches of the tree,
- how to print a selected entry

The Event Class

Event is a descendent of TObject. As such it inherits the data members of TObject and its methods such as Dump() and Inspect() and Write(). Also, because it inherits from TObject it can be a member of a collection.

To summarize, the advantages of inheriting from a TObject are:

- Inherit the Write, Inspect, and Dump methods
- Enables a class to be a member of a ROOT collection
- Enables RTTI

Below is the list of the Event data members. It contains a character array, several integers, a floating point number, and an EventHeader object. The EventHeader class is described in the following paragraph.

The string "->" in the comment field of the members *fTracks and *fH instructs the automatic Streamer to assume that the objects *fTracks and *fH are never null pointers and that fTracks->Streamer can be used instead of the more time consuming form R__b << fTracks.

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

The EventHeader Class

The EventHeader class (also defined in Event.h) does not inherit from TObject. Beginning with ROOT 3.0, an object can be placed on a branch even though it does not inherit from TObject. In previous releases branches were restricted to objects inheriting from the TObject. However, it has always been possible to write a class not inheriting from TObject to a tree by encapsulating it in a TObject descendent class as is the case in EventHeader and Event.

```c
class EventHeader {
private:
  Int_t fEvtNum;
  Int_t fRun;
  Int_t fDate;
  // ... list of methods
  // ClassDef(EventHeader,1)  //Event Header
};
```

The Track Class

The Track class descends from TObject since tracks are in a TClonesArray (i.e. a ROOT collection class) and contains a selection of basic types and an array of vertices. It’s TObject inheritance, enables Track to be in a collection, and in Event is a TClonesArray of Tracks.

```c
class Track : public TObject {
private:
  Float_t fPx;       //X component of the momentum
  Float_t fPy;       //Y component of the momentum
  Float_t fPz;       //Z component of the momentum
  Float_t fRandom;   //A random track quantity
  Float_t fMass2;    //The mass square of this particle
  Float_t fBx;       //X intercept at the vertex
  Float_t fBy;       //Y intercept at the vertex
  Float_t fMeanCharge;  //Mean charge deposition of all hits
  Float_t fXfirst;   //X coordinate of the first point
  Float_t fXlast;    //X coordinate of the last point
  Float_t fYfirst;   //Y coordinate of the first point
  Float_t fYlast;    //Y coordinate of the last point
  Float_t fZfirst;   //Z coordinate of the first point
  Float_t fZlast;    //Z coordinate of the last point
  Float_t fCharge;   //Charge of this track
  Int_t fNPoint;     //Number of points for this track
  Short_t fValid;    //Validity criterion
  // ... method definitions ...
  // ClassDef(Track,1)  //A track segment
};
```
Writing the Tree

We create a simple tree with two branches both holding Event objects. One is split and the other is not. We also create a pointer to an Event object (event).

```cpp
void tree4w()
{
    // check to see if the event class is in the dictionary
    // if it is not load the definition in libEvent.so
    if (!TClassTable::GetDict("Event")) {
        gSystem->Load("$ROOTSYS/test/libEvent.so");
    }

    // create a tree file tree4.root
    TFile f("tree4.root","RECREATE");

    // Create a ROOT Tree
    TTree t4("t4","A Tree with Events");

    // Create a pointer to an Event object
    Event *event = new Event();

    // Create two branches, split one.
    t4.Branch("event_branch", "Event", &event,16000,2);
    t4.Branch("event_not_split", "Event", &event,16000,0);

    // a local variable for the event type
    char etype[20];

    // Fill the tree
    for (Int_t ev = 0; ev <100; ev++) {
        Float_t sigmat, sigmas;
        gRandom->Rannor(sigmat,sigmas);
        Int_t ntrack   = Int_t(600 + 600 *sigmat/120.);
        Float_t random = gRandom->Rndm(1);
        sprintf(etype,"type%d",ev%5);
        event->SetType(etype);
        event->SetHeader(ev, 200, 960312, random);
        event->SetNseg(Int_t(10*ntrack+20*sigmas));
        event->SetNvertex(Int_t(1+20*gRandom->Rndm()));
        event->SetFlag(UInt_t(random+0.5));
        event->SetTemperature(random+20.);
        for(UChar_t m = 0; m < 10; m++) {
            event->SetMeasure(m, Int_t(gRandom->Gaus(m,m+1)));
        }
        // fill the matrix
        for(UChar_t i0 = 0; i0 < 4; i0++) {
            for(UChar_t i1 = 0; i1 < 4; i1++) {
                event->SetMatrix(i0,i1,gRandom->Gaus(i0*10,i1*10));
            }
        }
        //.. continued
    }
}
```

Reading the Tree

First, we check if the shared library with the class definitions is loaded. If not we load it. Then we read two branches, one for the number of tracks and one for the entire event. We check the number of tracks first, and if it meets our condition we read the entire event. We show the first entry that meets the condition.

```cpp
void tree4r()
{
    // check to see if the event class is in the dictionary
    // if it is not load the definition in libEvent.so
    if (!TClassTable::GetDict("Event")) {
        gSystem->Load("$ROOTSYS/test/libEvent.so");
    }

    // read the tree generated with tree4w
    TFile *f = new TFile("tree4.root");
    TTree *t4 = (TTree*)f->Get("t4");

    // create a pointer to an event object. This will be used
    // to read the branch values.
    Event *event = new Event();

    // get two branches and set the branch address
    TBranch *btrack = t4->GetBranch("fNtrack");
    TBranch *branch = t4->GetBranch("event_split");
    branch->SetAddress(&event);
    Int_t nevent = t4->GetEntries();
    Int_t nselected = 0;
    Int_t nb = 0;
    //.. continued ...
```
for (Int_t i=0;i<nevent;i++) {
    //read branch "fNtrack" only
    bntrack->GetEntry(i);
    
    // reject events with more than 587 tracks
    if (event->GetNtrack() > 587) continue;
    
    // read complete accepted event in memory
    nb += t4->GetEntry(i);
    nselected++;
    
    // print the first accepted event
    if (nselected == 1) t4->Show();
    
    // clear tracks array
    event->Clear();
}

if (gROOT->IsBatch()) return;
new TBrowser();
t4->StartViewer();
}

Now, let's see what the tree looks like in the tree viewer.

The TClonesArray of tracks fTracks is also split because we set the split level to 2.

The output on the command line is the result of tree4->Show. It shows the first entry with more than 587 tracks:

======> EVENT:26
    event_split =
    fUniqueID   = 0
    fBits       = 50331648
    fType[20]   = 116 121 112 101 49 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
    fNtrack     = 585
    fNseg       = 585
    fNvertex    = 17
    fFlag       = 0
    fTemperature= 20.044315
    fEvtHdr.fEvtNum = 26
    fEvtHdr.fRun   = 200
    fEvtHdr.fDate  = 960312
    fTracks      = 585
    fTracks.fUniqueID = 0, 0, 0, 0, 0, 0, 0, 0, 0, 0

Trees in Analysis

The methods TTree::Draw, TTree::MakeClass, and TTree::MakeSelector are available for data analysis using trees.

The TTree::Draw method is a powerful yet simple way to look and draw the trees contents. It enables you to plot a variable (a leaf) with just one line of code. However, the Draw method falls short once you want to look at each entry and design more sophisticated acceptance criteria for your analysis. For these cases, you can use TTree::MakeClass. It creates a class that loops over the trees entries one by one. You can then expand it to do the logic of your analysis.

The TTree::MakeSelector is the recommended method for ROOT data analysis. It is especially important for large data set in a parallel processing configuration where the analysis is distributed over several processors and you can specify which entries to send to each processors. With MakeClass the user has control over the event loop, with MakeSelector the tree is in control of the event loop.

Simple Analysis using TTree::Draw

We will use the tree in staff.root which was made by the macro in $ROOTSYS/tutorials/staff.C

First, open the file and lists its contents.

root [] TFile f ("staff.root")
root [] f.ls()
TFile** staff.root
TFile* staff.root
KEY: TTree tree;1 staff data from ascii file

We can see the TTree "tree" in the file. We will use it to experiment with the TTree::Draw method, so let's create a pointer to it:

root [] TFile f ("staff.root")
root [] f.ls()
TFile** staff.root
TFile* staff.root
root [] tree ;1 staff data from ascii file

Now, let's see what the tree looks like in the tree viewer.

You can see the two branches in the tree in the left panel: the event_branch is split and hence expands when clicked on. The other branch event_not_split is not expandable and we can not browse the data members.
CINT allows us to simply get the object by using it. Here we define a pointer to a TTree object and assign it the value of "tree", the TTree in the file. CINT looks for "tree" and returns it.

To show the different Draw options, we create a canvas with four sub-pads. We will use one sub-pad for each Draw command.

```
TCanvas *myCanvas = new TCanvas()
myCanvas->Divide(2,2)
```

We activate the first pad with the TCanvas::cd statement:

```
myCanvas->cd(1)
```

We then draw the variable cost:

```
MyTree->Draw("cost")
```

As you can see this call to TTree::Draw has only one parameter. It is a string containing the leaf name.

A histogram is automatically created as a result of a TTree::Draw. The style of the histogram is inherited from the TTree attributes and the current style (gStyle) is ignored. The TTree gets its attributes from the current TStyle at the time it was created. You can call the method TTree::UseCurrentStyle to change to the current style rather than the TTree style (see gStyle, see the Chapter Graphics and Graphic User Interfaces).

In this next segment we activate the second pad and draw a scatter plot variables:

```
myCanvas->cd(2)
```

This signature still only has one parameter, but it now has two dimensions separated by a colon ("x:y"). The item to be plotted can be an expression not just a simple variable. In general, this parameter is a string that contains up to three expressions, one for each dimension, separated by a colon ("e1:e2:e3"). A list of examples follows this introduction.

Using Selection with TTree:Draw

Change the active pad to 3, and add a selection to the list of parameters of the draw command.

```
myCanvas->cd(3)
MyTree->Draw("cost:age","nation == 3");
```

This will draw the cost vs. age for the entries where the nation is equal to 3. You can use any C++ operator, plus some functions defined in TFormula, in the selection parameter.

```
Selection = "weight *(boolean expression)"
```

If the Boolean expression evaluates to true, the histogram is filled with a weight. If the weight is not explicitly specified it is assumed to be 1.

For example, this selection will add 1 to the histogram if x is less than y and the square root of z is less than 3.2.

```
"x<y && sqrt(z)<3.2"
```

On the other hand, this selection will add x+y to the histogram if the square root of z is larger than 3.2.

```
"(x+y)*(sqrt(z)>3.2)"
```

The Draw method has its own parser, and it only looks in the current tree for variables. This means that any variable used in the selection must be defined in the tree. You cannot use an arbitrary global variable in the TTree::Draw method.

Using TCut Objects in TTree::Draw

The TTree::Draw method also accepts TCut objects. A TCut is a specialized string object used for TTree selections. A TCut object has a name and a title. It does not have any data members in addition to what it inherits from TNamed. It only adds a set of operators to do logical string concatenation. For example, assume:

```
TCut c1 = "x<1"
TCut c2 = "y<0"
```

then

```
cut1 && cut2
//result is the string "(x<1)&&(y<0)"
```

Operators =, +=, +, *, !, &&, || are overloaded, here are some examples:

```
root[] TCut c1 = "x<1"
root[] TCut c2 = "y<0"
root[] TCut c3 = c1 && c2
root[] MyTree.Draw("x", c1)
root[] MyTree.Draw("x", c1 || "x>0")
root[] MyTree.Draw("x", c1 && c2)
root[] MyTree.Draw("x", *(x+y) * (c1 && c2))
```
Accessing the Histogram in Batch Mode

The TTree::Draw method creates a histogram called htemp and puts it on the active pad.

In a batch program, the histogram htemp created by default, is reachable from the current pad.

```
// draw the histogram
nt->Draw("x", "cuts");
// get the histogram from the current pad
TH1F htemp = (TH1F*) gPad->GetPrimitive("htemp");
// now we have full use of the histogram
htemp->GetEntries();
```

If you pipe the result of the TTree::Draw into a histogram, the histogram is also available in the current directory. You can do:

```
// Draw the histogram and fill hnew with it
nt->Draw("x>>hnew","cuts");
// get hnew from the current directory
TH1F *hnew = (TH1F*)gDirectory->Get("hnew");
// or get hnew from the current Pad
TH1F *hnew = (TH1F*)gPad->GetPrimitive("hnew");
```

Superimposing two Histograms

When superimposing two 2-D histograms inside a script with TTree::Draw and using the "same" option, you will need to update the pad between Draw commands.

```
// superimpose two 2D scatter plots
{
// Create a 2D histogram and fill it with random numbers
TH2 *h2 = new TH2D("h2","2D histo",100,0,70,100,0,20000);
for (Int_t i = 0; i < 10000; i++)
h2->Fill(gRandom->Gaus(40,10),gRandom->Gaus(10000,3000));
// set the color to differentiate it visually
h2->SetMarkerColor(kGreen);
h2->Draw();
// Open the example file and get the tree
TFile f("staff.root");
TTree *myTree = (TTree*)f.Get("tree");
// the update is needed for the next draw command to // work properly
gPad->Update();
myTree->Draw("cost:age", ",","same");
}
```

Using Draw Options in TTree::Draw

The next parameter is the draw option for the histogram:

```
root [] myCanvas->cd(4) 
root [] MyTree->Draw("cost:age","nation == 3", "surf2");
```

The draw options are the same as for TH1::Draw, and they are listed in the section: Draw Options in the chapter on Histograms. In addition to the draw options defined in TH1, there are three more.

The 'prof' and 'profs' that draw a profile histogram (TProfile) rather than a regular 2D histogram (TH2D) from an expression with two variables. If the expression has three variables, a TProfile2D is generated.

The 'prof' generates a TProfile with error on the spread. The 'prof' option generates a TProfile with error on the mean.

The "gof" option suppresses generating the graphics.

You can combine the draw options in a list separated by commas. After typing the lines above, you should now have a canvas that looks like this.

Setting the Range in TTree::Draw

There are two more optional parameters to the TTree::Draw method: one is the number of entries and the second one is the entry to start with. For example this command draws 1000 entries starting with entry 100:

```
myTree->Draw("cost:age", "","same");
```

In this example, h2->Draw is only adding the object h2 to the pad's list of primitives. It does not paint the object on the screen. However, TTree::Draw when called with option 'same' gets the current pad coordinates to build an intermediate histogram with the right limits. Since nothing has been painted in the pad yet, the pad limits have not been computed. Calling pad->Update forces the painting of the pad and allows TTree::Draw to compute the right limits for the intermediate histogram.

TTree::Draw Examples

The examples below use the Event.root file generated by the $ROOTSYS/test/Event executable and the Event, Track, and EventHeader class definitions are in $ROOTSYS/test/Event.h.

The commands have been tested on the split levels 0, 1, and 9. Each command is numbered and referenced by the explanations immediately following the examples.
// Data members and methods
1. tree->Draw("fNtrack");
2. tree->Draw("event.GetNtrack()");
3. tree->Draw("GetNtrack()");
4. tree->Draw("fH.fXaxis.fXmax");
5. tree->Draw("fH.fXaxis.GetXmax()");
6. tree->Draw("GetNtrack()");
7. tree->Draw("GetHistogram().GetXaxis().GetXmax()");

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

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

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

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

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

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

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

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

// Alphanumeric bin histogram
37. tree->Draw("fNtrack")
// Where fNtrack and Division is a char* intended to be used
// as a string.
38. tree->Draw("MyByte + 0")
// Where MyByte is a char* intended to be used as a byte.

Explanations:
1. tree->Draw("fNtrack")
   Fills the histogram with the number of tracks for each entry. fNtrack
   is a member of event.
2. tree->Draw("event.GetNtrack()")
   Same as case 1, but use the method of event to get the number of
   tracks. When using a method, you can include parameters for the method
   as long as the parameters are literals.
3. tree->Draw("GetNtrack()")
   Same as case 2, the object of the method is not specified. The command
   uses the first instance of the GetNtrack method found in the objects
   stored in the tree. We recommend using this shortcut only if the method
   name is unique.
4. tree->Draw("fH.fXaxis.fXmax")
   Draw the data member of a data member. In the tree, each entry has a
   histogram. This command draws the maximum value of the X-axis for
   each histogram.
5. tree->Draw("fH.fXaxis.GetXmax()")
   Same as case 4, but use the method of a data member.
6. tree->Draw("fH.GetXaxis().GetXmax()")
   Same as case 4, a data member of a data member retrieved by a
   method.
7. tree->Draw("GetHistogram().GetXaxis().GetXmax()")
   Same as case 4, using only methods.
8. tree->Draw("fH.fXaxis.fXmax")
   Use data members in the expression and in the selection parameter to
   plot fPx or all tracks in every 10th entry. Since fTracks is a
   TClonesArray of Tracks, there will be d values of fPx for each entry.
9. tree->Draw("fPx","fEvtHdr.fEvtNum%10 == 0")
   Same as case 8, use the name of the data member directly.
10. tree->Draw("fMatrix")
When the index of the array is left out or when empty brackets are used [], all values of the array are selected.

Draw all values of fMatrix for each entry in the tree. If fMatrix is defined as: Float_t fMatrix[4][4], all 16 values are used for each entry.

11. tree->Draw("fMatrix[ ][ ]");
   The same as case 10, all values of fMatrix are drawn for each entry.

12. tree->Draw("fMatrix[2][2]");
   The single element at fMatrix[2][2] is drawn for each entry.

13. tree->Draw("fMatrix[ ][0][ ]");
    Four elements of fMatrix are used: fMatrix[1][0], fMatrix[2][0], fMatrix[3][0], fMatrix[4][0].

14. tree->Draw("fMatrix[1][ ]");
    Four elements of fMatrix are used: fMatrix[1][0], fMatrix[1][1], fMatrix[1][2], fMatrix[1][3], fMatrix[1][4].

15. tree->Draw("fMatrix - fVertex");
    With two arrays and unspecified element numbers, the number of selected values is the minimum of the first dimension times the minimum of the second dimension. In this case fVertex is also a two dimensional array since it is a data member of the tracks array. If fVertex is defined in the track class as: Float_t *fVertex[3], it has ftracks x 3 elements. fMatrix has 4 x 4 element. This case, draws 4 (the lesser of ftracks and 4) times 3 (the lesser of 4 and 3), meaning 12 elements per entry. The selected values for each entry are:

   fMatrix[0][0] - fVertex[0][0]
fMatrix[0][1] - fVertex[0][1]
fMatrix[0][2] - fVertex[0][2]
fMatrix[0][3] - fVertex[0][3]
fMatrix[1][0] - fVertex[1][0]
fMatrix[1][1] - fVertex[1][1]
fMatrix[1][2] - fVertex[1][2]
fMatrix[1][3] - fVertex[1][3]
fMatrix[2][0] - fVertex[2][0]
fMatrix[2][1] - fVertex[2][1]
fMatrix[2][3] - fVertex[2][3]
fMatrix[3][0] - fVertex[3][0]
fMatrix[3][1] - fVertex[3][1]
fMatrix[3][2] - fVertex[3][2]
fMatrix[3][3] - fVertex[3][3]

16. tree->Draw("fMatrix[2][1] - fVertex[5][1]");
    This command selects one value per entry.

17. tree->Draw("fMatrix[1][1] - fVertex[5][1]");
    The first dimension of the array is taken by the fMatrix.

   fMatrix[0][1] - fVertex[5][1]
fMatrix[1][1] - fVertex[5][1]
fMatrix[2][1] - fVertex[5][1]
fMatrix[3][1] - fVertex[5][1]

18. tree->Draw("fMatrix[2][ ] - fVertex[5][ ]");
    The first dimension minimum is 2, and the second dimension minimum is 3 (from fVertex). Three values are selected from each entry:

   fMatrix[2][0] - fVertex[5][0]
fMatrix[2][1] - fVertex[5][1]
fMatrix[2][2] - fVertex[5][2]

19. tree->Draw("fMatrix[ ][2] - fVertex[ ][1]");
    This is similar to case 18. Four values are selected from each entry:

   fMatrix[0][2] - fVertex[0][1]
fMatrix[1][2] - fVertex[1][1]
fMatrix[2][2] - fVertex[2][1]
fMatrix[3][2] - fVertex[3][1]

20. tree->Draw("fMatrix[ ][2] - fVertex[ ][ ]");
    This is similar to case 19. Twelve values are selected (4x3) from each entry:

   fMatrix[0][2] - fVertex[0][0]
fMatrix[0][2] - fVertex[0][1]
fMatrix[0][2] - fVertex[0][2]
fMatrix[1][2] - fVertex[1][0]
fMatrix[1][2] - fVertex[1][1]
fMatrix[1][2] - fVertex[1][2]
fMatrix[2][2] - fVertex[2][0]
fMatrix[2][2] - fVertex[2][1]
fMatrix[3][2] - fVertex[3][0]
fMatrix[3][2] - fVertex[3][1]
fMatrix[3][2] - fVertex[3][2]

21. tree->Draw("fMatrix[ ][ ] - fVertex[ ][ ]");
    This is the same as case 15. The first dimension minimum is 4 (from fMatrix), and the second dimension minimum is 3 (from fVertex). Twelve values are selected from each entry.

22. tree->Draw("fcClosestDistance");
    This event data member fClosestDistance is a variable length array: Float_t *fClosestDistance; [//[fNvertex]]. This command selects all elements, but the number per entry depends on the number of vertices of that entry.

23. tree->Draw("fcClosestDistance,fNvertex/2");
    With this command the element at fNvertex/2 of the fClosestDistance array is selected. Only one per entry is selected.

24. tree->Draw("sqrt(fPx*fPx + fPy*fPy + fPz*fPz)");
    This command shows the use of a mathematical expression. It draws the square root of the sum of the product.

25. tree->Draw("fEvtHdr.fEvtNum","fType=="type1"");
    You can compare strings, using the symbols == and !=, in the first two parameters of the Draw command (TFtreeFormula). In this case, the event number for 'type1' events is plotted.

26. tree->Draw("fEvtHdr.fEvtNum","strstr(fType,\"type1\")");
    To compare strings, you can also use strstr. In this case, events having a '1' in fType are selected.

27. tree->Draw("fTracks.fPoints");
    If fPoints is a data member of the Track class declared as: Int_t *fPoints; [fNpoint]

   Int_t *fPoints; [fNpoint]
The size of the array `fPoints` varies with each track of each event. This command draws all the value in the `fPoints` arrays.

28. `tree->Draw("fTracks.fPoints[0]", "fTracks.fPoints[][fAvgPoints]")`;

When `fAvgPoints` is a data member of the `Event` class, this example selects:

- `fTracks[0].fPoints[0] - fTracks[0].fPoint[fAvgPoints]`
- `fTracks[0].fPoints[1] - fTracks[0].fPoint[fAvgPoints]`
- `fTracks[0].fPoints[2] - fTracks[0].fPoint[fAvgPoints]`
- `fTracks[0].fPoints[3] - fTracks[0].fPoint[fAvgPoints]`
- `fTracks[0].fPoints[4] - fTracks[0].fPoint[fAvgPoints]`
- ... `fTracks[0].fPoints[max0] - fTracks[0].fPoint[fAvgPoints]`
- `fTracks[1].fPoints[0] - fTracks[1].fPoint[fAvgPoints]`
- ... `fTracks[1].fPoints[max1] - fTracks[1].fPoint[fAvgPoints]`
- ... `fTracks[fNtrack-1].fPoints[0] - fTracks[fNtrack-1].fPoint[fAvgPoints]`
- ... `fTracks[fNtrack-1].fPoints[maxn] - fTracks[fNtrack-1].fPoint[fAvgPoints]`

Where `max0`, `max1`, ... `max n`, is the size of the `fPoints` array for the respective track.

29. `tree->Draw("fTracks.fPoints[2]", "(fNvertex>10) && (fNseg<=6000)")`

You can use bit patterns (|, &<<) or Boolean operation.

30. `tree->Draw("fPx","(fBx>.4) || (fBy<=-.4)")`

31. `tree->Draw("MyChar +0")`

If you want to plot a `char` variable as a byte rather than a string, you can use the syntax above.
Creating an Event List

The TTree::Draw method can also be used to build a list of the entries. When the first argument is preceded by ">>" ROOT knows that this command is not intended to draw anything, but to save the entries in a list with the name given by the first argument. The resulting list is a TEventList, and is added to the objects in the current directory.

For example, to create a TEventList of all entries with more than 600 tracks:

```c
root [] TFile *f = new TFile("Event.root")
root [] T->Draw(">> myList", " fNtrack > 600")
```

This list contains the entry number of all entries with more than 600 tracks. To see the entry numbers use the Print("all") command.

```c
root [] myList->Print("all")
```

When using the ">>" whatever was in the TEventList is overwritten. The TEventList can be grown by using the ">>+") syntax.

For example to add the entries, with exactly 600 tracks:

```c
root [] T->Draw(">>+ myList", " fNtrack == 600")
```

This command does not add any new entries to the list because all entries with more than 610 tracks have already been found by the previous command for entries with more than 600 tracks.

Using an Event List

The TEventList can be used to limit the TTree to the events in the list. The SetEventList method tells the tree to use the event list and hence limits all subsequent TTree methods to the entries in the list. In this example, we created a list with all entries with more than 600 tracks and then set it so the Tree will use this list. To reset the TTree to use all events use SetEventList(0).

1) Let's look at an example. First, open the file and draw the fNtrack.

```c
root [] TFile *f = new TFile("Event.root")
root [] T->Draw("fNtrack ")
```

2) Now, put the entries with over 600 tracks into a TEventList called myList. We get the list from the current directory and assign it to a variable l.

```c
root [] TFile *f = new TFile("Event.root")
root [] T->Draw("fNtrack ")
```

Filling a Histogram

The TTree::Draw method can also be used to fill a specific histogram. The syntax is:

```c
root [] TFile *f = new TFile("Event.root")
root [] T->Draw("fNtrack >> myHisto")
root [] myHisto->Print()
```

As we can see, this created a TH1 called myHisto. If you want to append more entries to the histogram, you can use this syntax:

```c
root [] T->Draw("fNtrack >>+ myHisto")
```

If you do not create a histogram ahead of time, ROOT will create one at the time of the Draw command (as is the case above). If you would like to draw the variable into a specific histogram where you, for example, set the range and bin number, you can define the histogram ahead of time and use it in the Draw command. The histogram has to be in the same directory as the tree.

```c
TH1 *h1 = new TH1("h1","h1",50, 0., 150.);
root [] T->Draw("fNtrack>> h1");
```
the method TTree::UseCurrentStyle to change the histogram to use the current style (gStyle, see the Chapter Graphics and Graphic User Interfaces).

The binning of the newly created histogram can be specified in two ways. You can set a defaults in the .rootrc and/or you can add the binning information in the TTree::Draw command.

To set number of bins default for the 1-d, 2-d, 3-d histograms can be specified in the .rootrc file via the environment variables, e.g.:

```
# default binnings
Hist.Binning.1D.x: 100
Hist.Binning.2D.x: 40
Hist.Binning.2D.y: 40
Hist.Binning.2D.Prof: 100
Hist.Binning.3D.x: 20
Hist.Binning.3D.y: 20
Hist.Binning.3D.z: 20
Hist.Binning.3D.Profx: 100
Hist.Binning.3D.Profy: 100
```

To set the number of bins for a specific histogram when using TTree::Draw, add up to nine numbers following the histogram name. The numbers meaning is:

1 - bins in x-direction
2 - lower limit in x-direction
3 - upper limit in x-direction
4-6 same for y-direction
7-9 same for z-direction

When a bin number is specified, the value becomes the default. Any of the numbers can be skipped. For example:

```
tree.Draw("sqrt(x)>>hsqrt(500,10,20)";
// plot sqrt(x) between 10 and 20 using 500 bins
```

When the name is followed by binning information, appending the histogram with a "+", will not reset hsqrt, but will continue to fill it.

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

This works for 1-D, 2-D and 3-D histograms.

### Projecting a Histogram

If you would like to fill a histogram, but not draw it you can use the TTree::Project() method.

### Making a Profile Histogram

In case of a two dimensional expression, you can generate a TProfile histogram instead of a two dimensional histogram by specifying the 'prof' or 'profns' option. The prof option is automatically selected when the output is redirected into a TProfile. For example y=x>pf where pf is an existing TProfile histogram.

### Tree Information

Once we have drawn a tree, we can get information about the tree. These are the methods used to get information from a drawn tree:

- GetSelectedRows: Returns the number of entries accepted by the selection expression. In case where no selection was specified, it returns the number of entries processed.
- GetV1: Returns a pointer to the float array of the first variable.
- GetV2: Returns a pointer to the float array of second variable
- GetV3: Returns a pointer to the float array of third variable.
- GetW: Returns a pointer to the float array of Weights where the weight equals the result of the selection expression.

To read the drawn values of fNtrack into an array, and loop through the entries follow the lines below. First, open the file and draw the fNtrack variable:

```
root [] T->Project("quietHisto","fNtrack")
```

Then declare a pointer to a float and use the GetV1 method to retrieve the first dimension of the tree. In this example we only drew one dimension (fNtrack) if we had drawn two, we could use GetV2 to get the second one.

```
root [] Float_t *a
root [] a = T->GetV1()
```

Loop through the first 10 entries and print the values of fNtrack:

```
root [] for (int i = 0; i < 10; i++) cout<<a[i]<< " ";
root [] cout << endl // need an endl to see the values
```

By default, TTree::Draw creates these arrays with fEstimate words where fEstimate can be set via TTree::SetEstimate. If you have more entries than fEstimate only the first fEstimate selected entries will be stored in the arrays. The arrays are used as buffers. When fEstimate entries have been processed, ROOT scans the buffers to compute the minimum and maximum of each coordinate and creates the corresponding histograms.

You can use these lines to read all entries into these arrays:
Using TTree::MakeClass

The TTree::Draw method is convenient and easy to use, however it fails short if you need to do some programming with the variable.

For example, for plotting the masses of all oppositely charged pairs of tracks, you would need to write a program that loops over all events, finds all pairs of tracks, and calculates the required quantities. We have shown how to retrieve the data arrays from the branches of the tree in the previous section, and you could just write that program from scratch. Since this is a very common task, ROOT provides a utility that generates a skeleton class designed to loop over the entries of the tree. This is the TTree::MakeClass method.

We will now go through the steps of using MakeClass with a simplified example. The methods used here obviously work for much more complex event loop calculations.

These are our assumptions:

- We would like to do selective plotting and loop through each entry of the tree and tracks. We chose a simple example: we want to plot $f_{x,y}$ of the first 100 tracks of each entry.
- We have a ROOT tree with a branch for each data member in the "Event" object. To build this file and tree follow the instructions on how to build the examples in $ROOTSYS/test.

Execute Event and instruct it to split the object with this command (from the Unix command line).

```
$ROOTSYS/test/Event 400 1 2 1
```

This creates an Event.root file with 400 events, compressed, split, and filled. See $ROOTSYS/test/MainEvent.Cxx for more info.

The person who designed the tree makes a shared library available to you, which defines the classes needed. In this case, the classes are Event, EventHeader, and Track and they are defined in the shared library libEvent.so. The designer also gives you the Event.h file to see the definition of the classes. You can locate Event.h in $ROOTSYS/test, and if you have not yet built libEvent.so, please see the instructions of how to build it. If you have already built it, you can now use it again.

Creating a Class with MakeClass

First, we load the shared library and open Event.root.

```
root [] .L libEvent.so
root [] TFile *f = new TFile("Event.root");
root [] f->ls();
TFile** Event.root TTree benchmark ROOT file
TFile* Event.root TTree benchmark ROOT file
KEY: T1PF htime;1 Real-Time to write versus time
KEY: TTree T;1 An example of a TTree root
```

We can see there is a tree "T", and just to verify that we are working with the correct one, we print the tree, which will show us the header and branches.

```
root [] T->Print();
```

From the output of print we can see that the tree has one branch for each data member of Event, Track, and EventHeader.

Now we can use TTree::MakeClass on our tree "T". MakeClass takes one parameter, a string containing the name of the class to be made.

In the command below, the name of our class will be "MyClass".

```
root [] T->MakeClass("MyClass")
```

CINT informs us that it has created two files. MyClass.h, which contains the class definition and MyClass.C, which contains the MyClass::Loop method. MyClass has more methods than just Loop. The other methods are: a constructor, a destructor, GetEntry, LoadTree, Notify, and Show. The implementations of these methods are in the .h file. This division of methods was done intentionally. The .C file is kept as short as possible, and contains only code that is intended for you to customize. The .h file contains all the other methods.

It is clear that you want to be as independent as possible of the header file (i.e. MyClass.h) generated by MakeClass.

The solution is to implement a derived class, for example MyRealClass deriving from MyClass such that a change in your Tree or regeneration of MyClass does not force you to change MyRealClass.h.

You can imagine deriving several classes from MyClass.h, each with a specific algorithm.

To start with, it helps to understand both files, so lets start with MyClass.h and the class definition:
```cpp
class MyClass {
public:
// pointer to the analyzed TTree or TChain
TTree *fChain;
// current Tree number in a TChain
Int_t fCurrent;

// Declaration of leaves types
// Declaration of leaves types
UInt_t fUniqueID;
UInt_t fBits;
Char_t fType[20];
Int_t fNtrack;
Int_t fNseg;
Int_t fNvertex;
Float_t fTemperature;
Int_t fEvtHdr_fEvtNum;

// List of branches
TBranch *b_fUniqueID;
TBranch *b_fBits;
TBranch *b_fType;
TBranch *b_fNtrack;
TBranch *b_fNseg;
TBranch *b_fNvertex;
TBranch *b_fTemperature;
TBranch *b_fEvtHdr_fEvtNum;

MyClass(TTree *tree=0);
~MyClass();
Int_t Cut(Int_t entry);
Int_t GetEntry(Int_t entry);
Int_t LoadTree(Int_t entry);
void Init(TTree *tree);
void Loop();
Bool_t Notify();
void Show(Int_t entry = -1);
};
```

We can see data members in the generated class. The first data member is `fChain`. Once this class is instantiated, `fChain` will point to the original tree or chain this class was made from. In our case, this is "T" in "Event.root". If the class is instantiated with a tree as a parameter to the constructor, `fChain` will point to the tree named in the parameter.

Next is `fCurrent`, which is also a pointer to the current tree/chain. Its role is only relevant when we have multiple trees chained together in a TChain.

The class definition shows us that this tree has one branch and one leaf per data member. The methods of the class are:

- **MyClass(TTree *tree=0)**: This constructor has an optional tree for a parameter. If you pass a tree, `MyClass` will use it rather than the tree from which it was created.

- **void Init(TTree *tree)**: This is called by the constructor to initialize the tree for reading. It associates each branch with the corresponding leaf data member.

- **-MyClass:** This is the destructor, nothing special.

- **Int_t GetEntry(Int_t entry):** This loads the class with the entry specified. Once you have executed GetEntry, the leaf data members in MyClass are set to the values of the entry. For example, GetEntry(12) loads the 13th event into the event data member of MyClass (note that the first entry is 0). GetEntry returns the number of bytes read from the file. In case the same entry is read twice, ROOT does not have to do any I/O. In this case GetEntry returns 1. It does not return 0, because many people assume a return of 0 means an error has occurred while reading.

- **Int_t LoadTree(Int_t entry) and void Notify():** These two methods are related to chains. LoadTree will load the tree containing the specified entry from a chain of trees. Notify is called by LoadTree to adjust the branch addresses.

- **void Loop():** This is the skeleton method that loops through each entry of the tree. This is interesting to us, because we will need to customize it for our analysis.

```cpp
void MyClass::Loop()
{
if (fChain == 0) return;
Int_t nentries = Int_t(fChain->GetEntries());

Int_t nbytes = 0, nb = 0;
for (Int_t jentry=0; jentry<nentries;jentry++) {
Int_t ientry = LoadTree(jentry);
// in case of a TChain, ientry is the entry number
// in the current file
nb = fChain->GetEntry(jentry);   nbytes += nb;
// if (Cut(ientry) < 0) continue;
}
}
```

At the beginning of the file are instructions about reading selected branches. They are not reprinted here, but please read them from your own file.

### Modifying MyClass::Loop

Let's continue with the goal of going through the first 100 tracks of each entry and plot `Px`. To do this we change the Loop method.

```cpp
void MyClass::Loop()
{
if (fChain == 0) return;
Int_t nentries = Int_t(fChain->GetEntries());

TH1F *myHisto = new TH1F("myHisto","fPx", 100, -5,5);
TH1F *smallHisto = new TH1F("small","fPx", 100, -5,5);

for (Int_t jentry=0; jentry<nentries;jentry++) {
Int_t ientry = LoadTree(jentry);
// in case of a TChain, ientry is the entry number
// in the current file
nb = fChain->GetEntry(jentry);   nbytes += nb;
// if (Cut(ientry) < 0) continue;
}
}
```

In the for-loop, we need to add another for-loop to go over all the tracks. In the outer for-loop, we get the entry and the number of tracks. In the inner for-loop, we fill the large histogram (myHisto) with all tracks and
the small histogram (smallHisto) with the track if it is in the first 100.

```c
for (Int_t jentry=0; jentry<nentries; jentry++) {
    GetEntry(jentry);
    for (Int_t j = 0; j < 100; j++){
        myHisto->Fill(fTracks_fPx[j]);
        if (j < 100){
            smallHisto->Fill(fTracks_fPx[j]);
        }
    }
}
```

Outside of the for-loop, we draw both histograms on the same canvas.

```c
myHisto->Draw();
smallHisto->Draw("Same");
```

Save these changes to MyClass.C and start a fresh root session. We will now load MyClass and experiment with its methods.

### Loading MyClass

The first step is to load the library and the class file. Then we can instantiate a MyClass object.

```c
root [] .L libEvent.so
root [] .L MyClass.C
root [] MyClass m
```

Now we can get a specific entry and populate the event leaf. In the code snippet below, we get entry 0, and print the number of tracks (594). Then we get entry 1 and print the number of tracks (597).

```c
root [] m.GetEntry(0)
(int)57503
root [] m.fNtrack()
(Int_t)594
root [] m.GetEntry(1)
(int)48045
root [] m.fNtrack()
(Int_t)597
```

Now we can call the Loop method, which will build and display the two histograms.

```c
root [] m.Loop()
```

You should now see a canvas that looks like this.

To conclude the discussion on MakeClass let's lists the steps that got us here.

- Call TTree::MakeClass, which automatically creates a class to loop over the tree.
- Modify the MyClass::Loop() method in MyClass.C to fit your task.
- Load and instantiate MyClass, and run MyClass::Loop().

### Using TTree::MakeSelector

With a TTree we can make a selector and use it to process a limited set of entries. This is especially important in a parallel processing configuration where the analysis is distributed over several processors and we can specify which entries to send to each processors. The TTree::Process method is used to specify the selector and the entries.

Before we can use TTree::Process we need to make a selector. We can call the TTree::MakeSelector method. It creates two files similar to TTree::MakeClass. In the resulting files is a class that is a descendent of TSelector and implements the following methods:

- TSelector::Begin: This function is called every time a loop over the tree starts. This is a convenient place to create your histograms.
- TSelector::Notify(): This function is called at the first entry of a new tree in a chain.
- TSelector::ProcessCut: This function is called at the beginning of each entry to return a flag true if the entry must be analyzed.
- TSelector::ProcessFill: This function is called in the entry loop for all entries accepted by Select.
- TSelector::Terminate: This function is called at the end of a loop on a TTree. This is a convenient place to draw and fit your histograms.

The TSelector, unlike the resulting class from MakeClass, separates the processing into a ProcessCut and ProcessFill, so that we can limit reading the branches to the ones we need.

To create a selector call:

```c
root[] T->MakeSelector("MySelector");
```

Where T is the TTree and MySelector is the name of created class and the name of the .h and .C files.

The resulting TSelector is the argument to TTree::Process. The argument can be the file name or a pointer to the selector object.

```c
root[] T->Process("MySelector.C",**,1000,100);
```

This call will interpret the class defined in MySelector.C and process 1000 entries beginning with entry 100. The file name can be appended with a "**" or a "*++" to use ACLiC.

```c
root[] T->Process("MySelector.C;++",1000,100);
```

When appending a "**", the class will be compiled and dynamically loaded.

```c
root[] T->Process("MySelector.C;**,1000,100);
```

When appending a "*", the class will also be compiled and dynamically loaded. When it is called again, it recompiles only if the macro (MySelector.C) has changed since it was compiled last. If not it loads the existing library.

TTree::Process is aware of PROOF, ROOT's parallel processing facility. If PROOF is setup, it divides the processing amongst the slave CPUs.
Performance Benchmarks

The program $ROOTSYS/test/bench.cxx compares the I/O performance of STL vectors to the ROOT native TClonesArrays collection class. It creates trees with and without compression for the following cases:

- vector<THit>, vector<THit*>, TClonesArray(TObjHit) not split
- TClonesArray(TObjHit) split.

The graphs show the two columns on the right which represent the split and non-split TClonesArray, are significantly lower than the vectors. The most significant difference is when reading a file without compression. The file size with compression, write times with and without compression and the read times with and without compression all favor the TClonesArray.

Impact of Compression on I/O

This benchmark illustrates the pros and cons of the compression option. We recommend using compression when the time spent in I/O is small compared to the total processing time. In this case, if the I/O operation is increased by a factor of 5 it is still a small percentage of the total time and it may very well save a factor of 10 on disk space. On the other hand if the time spend on I/O is large, compression may slow down the program's performance.

The standard test program $ROOTSYS/test/Event was used in various configurations with 400 events. The data file contains a TTree. The program was invoked with:

```
Event 400 comp split
```

- 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.

These tests were run on Pentium III CPU with 650 Mhz.

<table>
<thead>
<tr>
<th>Event Parameters</th>
<th>File Size</th>
<th>Total Time to write (MB/sec)</th>
<th>Effective Time to write (MB/sec)</th>
<th>Total time to read All (MB/sec)</th>
<th>Total time to read Sample (MB/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp = 0  Split = 1</td>
<td>19.75 MB</td>
<td>6.84 s. (2.6 MB/s)</td>
<td>3.56 s. (5.4 MB/s)</td>
<td>0.79 s. (24.2 MB/s)</td>
<td>0.79 s. (24.2 MB/s)</td>
</tr>
<tr>
<td>Comp = 1  Split = 1</td>
<td>17.73 MB</td>
<td>6.44 s. (3.0 MB/s)</td>
<td>4.02 s. (4.8 MB/s)</td>
<td>0.90 s. (21.3 MB/s)</td>
<td>0.90 s. (21.3 MB/s)</td>
</tr>
<tr>
<td>Comp = 2  Split = 1</td>
<td>13.78 MB</td>
<td>11.34 s. (1.7 MB/s)</td>
<td>9.51 s. (2.0 MB/s)</td>
<td>2.17 s. (8.8 MB/s)</td>
<td>2.17 s. (8.8 MB/s)</td>
</tr>
</tbody>
</table>

The **Total Time** is the real time in seconds to run the program.

**Effective time** is 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 in the interactive version of ROOT. The **Total time to read All** is the real time reported by the execution of the script $ROOTSYS/test/eventa. We did not correct this time for the overhead coming from the interpreter itself.

The **Total time to read Sample** is the execution time of the script $ROOTSYS/test/eventb. This script 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.
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 the test programs on your architecture. The program Event will report the write performance. You can measure the read performance by executing the scripts eventa and eventb. The performance depends not only of the processor type, but also of the disk devices (local, NFS, AFS, etc.).

**Chains**

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

```cpp
TChain chain("T"); // name of the tree is the argument
chain.Add("file1.root");
chain.Add("file2.root");
chain.Add("file3.root");
```

The name of the TChain will be the same as the name of the tree, in this case it will be "T". Note that two objects can have the same name as long as they are not histograms in the same directory, because there, the histogram names are used to build a hash table.

The class TChain is derived from the class TTree. For example, to generate a histogram corresponding to the attribute "x" in tree "T" by processing sequentially the three files of this chain, we can use the TChain::Draw method.

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

When using a TChain, the branch address(es) must be set with:

```cpp
chain.SetBranchAddress("branch", &branch);
```

rather than:

```cpp
branch->SetAddress(...);
```

The second form returns the pointer to the branch of the current TTree in the chain, typically the first one. The information is lost when the next TTree is loaded.

The following statements 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.

```cpp
Event *event = new Event();
chain.SetBranchAddress("event", &event);
```

```cpp
for (Int_t i=0;i<nevent;i++) {
    chain.GetEvent(i);
    hnseg->Fill(event->GetNseg());
}
```

```cpp
hnseg->Draw();
```

**TChain::AddFriend**

A TChain has a list of friends similar to a tree (see TTree::AddFriend). You can add a friend to a chain with the TChain::AddFriend method, and you can retrieve the list of friends with TChain::GetListOfFriends.

This example has four chains each has 20 ROOT trees from 20 ROOT files.

```cpp
TChain ch("t"); // a chain with 20 trees from 20 files
TChain ch1("t1");
TChain ch2("t2");
TChain ch3("t3");
```

Now we can add the friends to the first chain.

```cpp
ch.AddFriend("t1")
ch.AddFriend("t2")
ch.AddFriend("t3")
```

The parameter is the name of friend chain (the name of a chain is always the name of the tree from which it was created).

The original chain has access to all variables in its friends. We can use the TChain::Draw method as if the values in the friends were in the original chain.
To specify the chain to use in the Draw method, use the syntax:

<chainname>.<branchname>.<varname>

If the variable name is enough to uniquely identify the variable, you can leave out the chain and/or branch name.

For example, this generates a 3-d scatter plot of variable "var" in the TChain ch versus variable v1 in TChain t1 versus variable v2 in TChain t2:

```
ch.Draw("var:t1.v1:t2.v2");
```

When a TChain::Draw is executed, an automatic call to TTree::AddFriend connects the trees in the chain. When a chain is deleted, its friend elements are also deleted.

The number of entries in the friend must be equal or greater to the number of entries of the original chain. If the friend has fewer entries a warning is given and the resulting histogram will have missing entries.

For additional information see TTree::AddFriends. A full example of a tree and friends is in Example #3 ($ROOTSYS/tutorials/tree3.c) in the Tree section above.
13 Adding a Class

The Role of TObject

The light-weight TObject class provides the default behavior and protocol for the objects in the ROOT system. Specifically, it is the primary interface to classes providing object I/O, error handling, inspection, introspection, and drawing. The interface to these service is via abstract classes.

Introspection, Reflection and Run Time Type Identification

Introspection, which is also referred to as reflection, or run time type identification (RTTI) is the ability of a class to reflect upon itself or to "look inside itself. ROOT implements reflection with the TClass class. It provides all the information about a class, a full description of data members and methods, including the comment field and the method parameter types. A class with the ClassDef macro, has the ability to obtain a TClass with the IsA method.

```
TClass *cl = obj->IsA();
```

which returns a TClass. In addition an object can directly get the class name and the base classes with:

```
const char* name = obj->ClassName();
```

which returns a character string containing the class name.

If the class is a descendent of TObject, you can check if an object inherits from a specific class, you can use the InheritsFrom method. This method returns kTrue if the object inherits from the specified class name or TClass.

```
Bool_t b = obj->InheritsFrom("TLine");
Bool_t b = obj->InheritsFrom(TLine::Class());
```

ROOT and CINT rely on reflection and the class dictionary to identify the type of a variable at run time.

With TObject inheritance come some methods that use Introspection to help you see the data in the object or class. For instance:

```
obj->Dump(); // lists all data members and their current value
obj->Inspect(); // opens a window to browser the data members at all levels
obj->DrawClass(); // draws the class inheritance tree
```

For an example of obj->Inspect see "Inspecting ROOT Objects" in the CINT chapter.

Collections

To store an object in a ROOT collection, it must be a descendent of TObject. This is convenient if you want to store objects of different classes in the same collection and execute the method of the same name on all members of the collection. For example the list of graphics primitives are in a ROOT collection called TList. When the canvas is drawn the Paint method is executed on the entire collection. Each member may be a different class, and if the Paint method is not implemented, TObject::Paint will be executed.

Input/Output

The TObject::Write method is the interface to the ROOT I/O system. It streams the object into a buffer using the Streamer method. It support cycle numbers and automatic schema evolution (see the chapter on I/O).

Paint/Draw

These two graphics methods are defaults, their implementation in TObject does not use the graphics subsystem. The TObject::Draw method is simply a call to AppendPad. The Paint method is empty. The default is provided so that one can call Paint in a collection.

GetDrawOption

This method returns the draw option that was used when the object was drawn on the canvas. This is especially relevant with histograms and graphs.

Clone/DrawClone

Two useful methods are Clone and DrawClone. The Clone method takes a snapshot of the object with the Streamer and creates a new object. The DrawClone method does the same thing and in addition draws the clone.

Browse

This method is called if the object is browse-able and is to be displayed in the object browser. For example the TBranch implementation of Browse, calls the Browse method for each branch. The TBranch::Browse method displays the name of each leaf. For the object's Browse method to be called, the IsFolder() method must be overridden to return true. This does not mean it has to be a folder, it just means that it is browse-able.
SavePrimitive
This method is called by a canvas on its list of primitives, when the canvas is saved as a script. The purpose of SavePrimitive is to save a primitive as a C++ statement(s). Most ROOT classes implement the SavePrimitive method. It is recommended that the SavePrimitive is implemented in user defined classes if it is to be drawn on a canvas. Such that the command TCanvas::SaveAs(Canvas.C) will preserve the user-class object in the resulting script.

GetObjectInfo
This method is called when displaying the event status in a canvas. To show the event status window, select the Options menu and the Event Status item. This method returns a string of information about the object at position (x, y). Every time the cursor moves, the object under the cursor executes the GetObjectInfo method. The string is then shown in the status bar.

There is a default implementation in TObject, but it is typically overridden for classes that can report peculiarities for different cursor positions (for example the bin contents in a TH1).

IsFolder
By default an object inheriting from TObject is not brows-able, because TObject::IsFolder() returns kFALSE. To make a class brows-able, the IsFolder method needs to be overridden to return kTRUE.

In general, this method returns kTRUE if the object contains brows-able objects (like containers or lists of other objects).

Bit Masks and Unique ID
A TObject descendant inherits two data members: fBits and fUniqueID.

fBits: This 32-bit data member is to be used with a bit mask to get information about the object. Bit 0–7 are reserved by TObject. The kMustClean, kCanDelete are used in TObject, these can be set by any object and should not be reused.

These are the bits used in TObject:

```cpp
enum EObjBits {
    kCanDelete = BIT(0), // if object in a list can be deleted
    kMustCleanup = BIT(3), // if object destructor must call RecursiveRemove()
    kCannotPick = BIT(6), // if object in a pad cannot be picked
    kInvalidObject = BIT(13), // object should not be used
};
```

The remaining 24 bits can be used by other classes. Make sure there is no overlap in any given hierarchy. For example TClass uses bit 12 and 13 kClassSaved and kIgnoreTObjectStreamer respectively.

The above bit 13 is set when an object could not be read from a ROOT file. It will check this bit and skip to the next object on the file.

The TObject constructor initializes the fBits to zero depending if the object is created on the stack or allocated on the heap. When the object is created on the stack, the kCanDelete bit is set to false to protect from deleting objects on the stack. Of the status word the high 8 bits are reserved for system usage and the low 24 bits are user settable.

fUniqueID: This data member can be used to give an object a unique identification number. It is initialized to zero by the TObject constructor. This data member is not used by ROOT.

These two data members are streamed out when writing an object to disk. If you do not use them you can save some space and time by specifying:

```cpp
MyClass::Class()->IgnoreTObjectStreamer()
```

This sets a bit in the TClass object.

If the file is compressed, the savings are minimal since most values are zero, however, it saves some space when the file is not compressed.

A call to IgnoreTObjectStreamer also prevents the creation of two additional branches when splitting the object. If left alone, two branches called fBits and fUniqueID will appear.

Motivation
If you want to integrate and use your classes with ROOT, to enjoy features like, extensive RTTI (Run Time Type Information) and ROOT object I/O and inspection, you have to add the following line to your class header files:

```cpp
ClassDef (ClassName,ClassVersionID) //The class title
```

For example in TLine.h we have:

```cpp
ClassDef (TLine,1) //A line segment
```

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 the section on Streamers in the Chapter Input/Output). Every time you change the data members 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:

```cpp
ClassImp(ClassName)
```

For example in TLine.cxx:

```cpp
ClassImp(TLine)
```

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 necessary to link your classes to the dictionary generated by CINT.
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.

### Template Support

ROOT provides `ClassDef` and `ClassImp` macros for classes with two and three template arguments. The macros are: `ClassDefT`, `ClassDef2T2`, `ClassDef3T2` and `ClassImpT`, `ClassImp2T`, `ClassImp3T`. `ClassDefT` is independent of the number of template arguments.

For templates the `ClassImp` must be in the header file. When you use templates in principle all the code is defined in the header. Then when a special "instantiation" is needed the compiler takes the code in the header and generates the real code (i.e. replaces the `T` by `int`, where `T` was the template argument). So for templated classes it is normal to put the `ClassImp` in the header.

Here is an example of a header `LinkDef` file:

```c
// in header file MyClass.h

// Error space for TList pointer will be lost
... public:
... T49Event() { fId = 0; fTrack = new TList; }

... if (!fTrack) fTrack = new TList;
```

### The Default Constructor

ROOT object I/O requires every class to have a default constructor. This default constructor is called whenever an object is being read from a ROOT database. Be sure that you don’t allocate any space for embedded pointer objects in the default constructor. This space will be lost (memory leak) while reading in the object. For example:

```c
... T49Event() { fId = 0; fTrack = 0; }
```
In the following example we walk through the steps necessary to generate a dictionary and I/O and inspect member functions.

Let's start with a TEvent class, which contains a collection of TTracks:

```c
#ifndef __TEvent__
#define __TEvent__
#include "TObject.h"

class TEvent : public TObject {
private:
    Int_t    fId;   // event sequential id
    Float_t  fTotalMom;  // total momentum
    TCollection *fTracks; // collection of tracks

public:
    TEvent() { fId = 0; fTracks = 0; }
    TEvent(Int_t id);
    ~TEvent();
    void    AddTrack(TTrack *t);
    Int_t   GetId() const { return fId; }
    Int_t   GetNoTracks() const;
    void    Print(Option_t *opt="");
    Float_t TotalMomentum();

    ClassDef (TEvent,1) //Simple event class
};

#endif
```

And the TTrack header:

```c
#ifndef __TTrack__
#define __TTrack__
#include "TObject.h"

class TEvent;

class TTrack : public TObject {
private:
    Int_t    fId;   //track sequential id
    TEvent  *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:
    TTrack() { fId = 0; fEvent = 0; fPx = fPy = fPz = 0; }
    TTrack(Int_t id, TEvent *ev, Float_t px, Float_t py, Float_t pz);
    Float_t  Momentum() const;
    TEvent  *GetEvent() const { return fEvent; }
    void     Print(Option_t *opt="");

    ClassDef (TTrack,1) //Simple track class
};
#endif
```

The things to notice in these header files are:

- The usage of the ClassDef macro
- The default constructors of the TEvent and TTrack classes
- Comments to describe the data members and the comment after the ClassDef macro to describe the class

These classes are intended for you to create an event object with a certain id, and then add tracks to it. The track objects have a pointer to their event. This shows that the I/O system correctly handles circular references.
Next, the implementation of these two classes. Event.cxx:

```c
#include <iostream.h>
#include "TOrdCollection.h"
#include "TEvent.h"
#include "TTrack.h"

ClassImp(TEvent)
...
...

and Track.cxx:

#include <iostream.h>
#include "TMath.h"
#include "Track.h"
#include "Event.h"

ClassImp(TTrack)
...
...
```

Now using rootcint we can generate the dictionary file.

Make sure you use a unique filename, because rootcint appends it to the name of static function (G__cpp_reset_tabableeventdict() and G__set_cpp_environmenteventdict()).

```
rootcint eventdict.cxx -c TEvent.h TTrack.h
```

Looking in the file eventdict.C we can see, besides the many member function calling stubs (used internally by the interpreter), the Streamer() and ShowMembers() methods for the two classes. Streamer() is used to stream an object from a TBuffer and ShowMembers() is used by the Dump() and Inspect() methods of TObject.

Here is the TEvent::Streamer method:

```c
void TEvent::Streamer(TBuffer &R__b)
{
  // Stream an object of class TEvent.
  if (R__b.IsReading()) {
    Version_t R__v = R__b.ReadVersion();
    TObject::Streamer(R__b);
    R__b >> fId;
    R__b >> fTotalMom;
    R__b >> fTracks;
  } else {
    R__b.WriteVersion(TEvent::IsA());
    TObject::Streamer(R__b);
    R__b << fId;
    R__b << fTotalMom;
    R__b << fTracks;
  }
}
```

The TBuffer class overloads the operator<<() and operator>>() for all basic types and for pointers to objects. These operators write and read from the buffer and take care of any needed byte swapping to make the buffer machine independent. During writing the TBuffer keeps track of the objects that have been written and multiple references to the same object are replaced by an index. In addition, the object’s class information is stored.

TEvent and TTracks need manual intervention. Cut and paste the generated Streamer() from the eventdict.C into the class’ source file and modify as needed (e.g. add counter for array of basic types) and disable the generation of the Streamer() when using the LinkDef.h file for next execution of rootcint.

In case you don’t want to read or write this class (no I/O) you can tell rootcint to generate a dummy Streamer() by changing this line in the source file:

```
ClassDef(TEvent,0)
```

If you want to prevent the generation of Streamer(), see the chapter "Adding a Class with a Shared Library" below.
Adding a Class with a Shared Library

Step 1:
Define your own class in SClass.h and implement it in SClass.cxx. You must provide a default constructor for your class.

```c++
#include <iostream.h>
#include "TObject.h"

class SClass : public TObject {
    private:
        Float_t fX;    // x position in centimeters
        Float_t fY;    // y position in centimeters
        Int_t fTempValue;    // temporary state value
    public:
        SClass()             { fX = fY = -1; }
        void Print() const;
        void SetX(float x) { fX = x; }
        void SetY(float y) { fY = y; }
        ClassDef (SClass, 1)
    }
}
```

Step 2:
Add a call to the ClassDef macro at the end of the class definition (i.e. in the SClass.h file).ClassDef(SClass,1).

Add a call to the ClassImp macro in the implementation file (SClass.cxx).ClassImp(SClass)

```c++
#include "SClass.h"
ClassImp (SClass);
void SClass::Print() const {
    cout << "fX = " << fX <<", fY = " << fY <<" endl;
}
```

You can add a class without using the ClassDef and ClassImp macros, however you will be limited. Specifically the object I/O features of ROOT will not be available to you for these classes (see the chapter "CINT the C++ Interpreter").

The `ShowMembers()` and `Streamer()` method, as well as the `>>` operator overloads, are implemented only if you use `ClassDef` and `ClassImp`.

See http://root.cern.ch/root/html/RTypes.h for the definition of `ClassDef` and `ClassImp`.

To exclude a data member from the Streamer, add a ! as the first character in the comments of the field:

```
Int_t fTempValue;    // temporary state value
```

Step 3:
The LinkDef.h file tells rootcint which classes to generate the method interface stubs.

```c++
#ifdef __CINT__
#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;
#pragma link C++ class SClass;
#endif
```

Three options can trail the class name:

• - : tells rootcint not to generate the `Streamer()` method for this class. This is necessary for those classes that need a customized `Streamer()` method.

```c++
#pragma link C++ class SClass-;  // no streamer
```

• !: tells rootcint not to generate the `operator>>(TBuffer &b, MyClass *&obj)` method for this class. This is necessary to be able to write pointers to objects of classes not inheriting from TObject.

```c++
#pragma link C++ class SClass!;  // no >> operator
```

• +: in ROOT version 1 and 2 tells rootcint to generate a `Streamer()` method with extra byte count information. This adds an integer to each object in the output buffer, but it allows for powerful error correction in case a `Streamer()` method is out of sync with data in the file. The + option is mutual exclusive with both the - and ! options.

```c++
#pragma link C++ class SClass+;  // add byte count
```

IMPORTANT NOTE: In ROOT Version 3, a "+" after the class name tells rootcint to use the new I/O system. The byte count check is always added.

```c++
#pragma link C++ class SClass++;  // add byte count
```

For information on Streamers see the section on Streamers in the Input/Output chapter.
To get help on rootcint type: rootcint -? on the UNIX command line.

The Order Matters

When using templated classes, the order of the pragma statements matters.
For example, here is a templated class Tmpl and a normal class Norm which holds a specialized instance of a Tmpl:

```
```
Then in Linkdef.h the pragma statements must be ordered by listing all specializations before any classes which need them:

```c
// Correct Linkdef.h ordering
#pragma link C++ class Tmpl<int>;   
#pragma link C++ class Norm;        
```

And not vice versa:

```c
// Bad Linkdef.h ordering
#pragma link C++ class Norm;        
#pragma link C++ class Tmpl<int>;   
```

In this later case, rootcint generates Norm::Streamer() which makes reference to Tmpl<int>::Streamer(). Then rootcint gets to process Tmpl<int>::Streamer() and generates a specialized Tmpl<int>::Streamer() function.

The problem is, when the compiler finds the first Tmpl<int>::Streamer(), it will instantiate it. But, later in the file it finds the specialized version that rootcint generated. This causes the error.

However, if the Linkdef.h order is reversed then rootcint can generate the specialized Tmpl<int>::Streamer() before it is needed (and thus never instantiated by the compiler).

**Step 4:** Compile the class using the Makefile

In the Makefile call rootcint to make the dictionary for the class. Call it SClassDict.cxx. The rootcint utility generates the Streamer(), TBuffer &operator>>() and ShowMembers() methods for ROOT classes.

For more information on rootcint follow this link:  

Also, see the $ROOTSYS/test directory Makefile, Event.cxx, and Event.h for an example.

---

**Adding a Class with ACLiC**

**Step 1. Define your class**

```c
#include "TObject.h"   
// define the ABC class and make it inherit class ABC : public TObject {   
  public:   
    Float_t a,b,c,p;   
    ABC():a(0),b(0),c(0),p(0){};   
  }   
}   
```

**Step 2: Load the ABC class in the script.**

```c
#include "TObject.h"   
// define the ABC class and make it inherit     
// from TObject so that we can write ABC to a ROOT file    
class ABC : public TObject {   
  public:   
    Float_t a,b,c,p;   
    ABC():a(0),b(0),c(0),p(0){};   
  }   
}   
```

**Step 3: Define the class for the cint dictionary**

```c
ClassDef (ABC,1)   
```
14 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 parameterized C++ collections or polymorphic collections before, some of this material will be review. However, much of this chapter covers aspects of collections specific to the ROOT system. When you have read this chapter, you will know

- How to create instances of collections
- The difference between lists, arrays, hash tables, 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 to access 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

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, a diagram editor might manage a collection of points and lines. A set of widgets for a graphical user interface can be placed in a collection. A geometrical model can be described by collections of shapes, materials and rotation matrices. Collections can themselves be placed in collections. Collections act as flexible alternatives to traditional data structures of computers science such as arrays, lists and trees.

General Characteristics

The ROOT collections are polymorphic containers that hold pointers to TObjectss, SO:

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

Collections are dynamic, they can grow in size as required.

Determining the Class of Contained Objects

Most containers may hold heterogeneous collections of objects and then it is left to the user to correctly cast the TObject pointer to the right class. Casting to the wrong class will give wrong results and may well crash the program! So the user has to be very careful. Often a container only contains one class of objects, but if it really contains a mixture, it is possible to ask each object about its class using the InheritsFrom() method.

For example if myObject is a TObject pointer:

```c
if (myObject->InheritsFrom("TParticle") { 
  printf("myObject is a TParticle\n"); 
}
```

As the name suggests, this test works even if the object is a subclass of TParticle. The member function IsA() can be used instead of InheritsFrom() to make the test exact. The InheritsFrom() and IsA() methods use the extensive Run Time Type Information (RTTI) available via the ROOT meta classes.

Types of Collections

The ROOT system implements the following basic types of collections: unordered collections, ordered collections and sorted collections. This picture shows the inheritance hierarchy for the primary collection classes. All primary collection classes derive from the abstract base class TCollection.
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 sequences are available:

- TList
- THashList
- TOrdCollection
- TObjectArray
- TClonesArray

The TOrdCollection, TObjectArray as well as the TClonesArray can be sorted using their Sort() member function (if the stored items are sort able). Ordered collections all derive from the abstract base class TSeqCollection.

Sorted Collections

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

- TSortedList
- TBtree

The stored items must be sort able.

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

Iterators: Processing a Collection

The concept of processing all the members of a collection is generic, i.e. independent of any specific representation of a collection. To process each object in a collection one needs some type of cursor that is initialized and then steps over each member of the collection in turn. Collection objects could provide this service but there is a snag: as there is only one collection object per collection there would only be one cursor. Instead, to permit the use of as many cursors as required, they are made separate classes called iterators. For each collection class there is an associated iterator class that knows how to sequentially retrieve each member in turn. The relationship between a collection and its iterator is very close and may require that the iterator has full access to the collection (i.e. it is a friend class). In general iterators will be used via the TIter wrapper class.

For example:

- TLList
- TListIter
- TObject
- TObjectIter

Foundation Classes

All collections are based on the fundamental classes: TCollection and TIterator. They are so generic that it is not possible to create objects from them; they are only used as base classes for other classes (i.e. they are abstract base classes).

TCollection

The TCollection class provides the basic protocol (i.e. the minimum set of member functions) that all collection classes have to implement. These include:

- Add() Adds another object to the collection.
- GetSize() Returns the number of objects in the collection.
- Clear() Clears out the collection, but does not delete the removed objects.
- Delete() Clears out and deletes the removed objects. This should only be used if the collection owns its objects (which is not normally the case).
- FindObject() Find an object given either its name or address.
- MakeIterator() Returns an iterator associated with the collection.
- Remove() Removes an object from the collection.

Coming back to the issue of object ownership. The code example below shows a class containing three lists, where the fTracks list is the owning collection and the other two lists are used to store a sub-set of the track objects. In the destructor of the class the Delete() method is called for the owning collection to delete correctly all its track objects.

To delete the objects in the container, do 'fTracks->Delete()'. To delete the container itself do 'delete fTracks'.

```cpp
class TEvent : public TObject {
private:
  TList *fTracks; //list of all tracks
  TList *fVertex1; //subset of tracks part of vertex1
  TList *fVertex2; //subset of tracks part of vertex2

  ...}

TEvent::~TEvent() {
  fTracks->Delete(); delete fTracks;
  delete fVertex1; delete fVertex2;
}
```

TIterator

The TIterator class defines the minimum set of member functions that all iterators must support. These include:

- Next() return the next member of the collection or 0 if no more members.
- Reset() reset the iterator so that Next() returns the first object.
**A Collectable Class**

By default, all objects of TObject derived classes can be stored in ROOT containers. However, the TObject class provides some member functions that allow you to tune the behavior of objects in containers. For example, by default two objects are considered equal if their pointers point to the same address. This might be too strict for some classes where equality is already achieved if some or all of the data members are equal. By overriding the following TObject member functions, you can change the behavior of objects in collections:

- **isEqual()** is used by the FindObject() collection method. By default, IsEqual() compares the two object pointers.
- **Compare()** returns -1, 0 or 1 depending if the object is smaller, equal or larger than the other object. By default, a TObject has not a valid Compare() method.
- **IsSortable()** returns true if the class is sort able (i.e. if it has a valid Compare() method). By default, a TObject is not sort able.
- **Hash()** returns a hash value. It needs to be implemented if an object has to be stored in a collection using a hashing technique, like THashTable, THashList and TMap. By default, Hash() returns the address of the object. It is essential to choose a good hash function.

The example below shows how to use and override these member functions:

```cpp
// TObjNum is a simple container for an integer.
class TObjNum : public TObject {
private:
    int  num;

public:
    TObjNum(int i = 0) : num(i) { }  
    ~TObjNum() { }  
    void     SetNum(int i)  { num = i; }
    int      GetNum() const  { return num; }
    void     Print(Option_t *) const 
    { printf("num = %d\n", num); }
    Bool_t   IsEqual(TObject *obj) const 
    { return num == ((TObjNum*)obj)->num; }
    Bool_t   IsSortable() const 
    { return kTRUE; }
    Int_t    Compare(TObject *obj) const 
    { if (num < ((TObjNum*)obj)->num) return -1;
      else if (num > ((TObjNum*)obj)->num) return 1;
      else return 0; }
    ULong_t  Hash() const { return num; }
};
```

**The TIter Generic Iterator**

As stated above, the TIterator class is abstract; it is not possible to create TIterator objects. However, it should be possible to write generic code to process all members of a collection so there is a need for a generic iterator object. A TIter object acts as generic iterator. It provides the same Next() and Reset() methods as TIterator although it has no idea how to support them! It works as follows:

- To create a TIter object its constructor must be passed an object that inherits from TCollection. The TIter constructor calls the MakeIterator() method of this collection to get the appropriate iterator object that inherits from TIterator.
- The Next() and Reset() methods of TIter simply call the Next() and Reset() methods of the iterator object.

So TIter simply acts as a wrapper for an object of a concrete class inheriting from TIterator.

To see this working in practice, consider the TObjArray collection. Its associated iterator is TObjArrayIter. Suppose myarray is a pointer to a TObjArray, i.e. TObjArray *myarray;

Which contains MyClass objects. To create a TIter object called myiter:

TIter myiter(myarray);

As shown in the diagram, this results in several methods being called:

1. The TIter constructor is passed a TObjArray
2. TIter asks embedded TCollection to make an iterator
3. TCollection asks TObjArray to make an iterator
4. TObjArray returns a TObjArrayIter.

As shown in the diagram, this results in several methods being called:

(1) The TIter constructor is passed a TObjArray
(2) TIter asks embedded TCollection to make an iterator
(3) TCollection asks TObjArray to make an iterator
(4) TObjArray returns a TObjArrayIter.
Now define a pointer for MyClass objects and set it to each member of the TObjArray:

```c
MyClass *myobject;
while ((myobject = (MyClass *) myiter.Next())) {
    // process myobject
}
```

The heart of this is the myiter.Next() expression which does the following:

1. The Next() method of the TIter object myiter is called
2. The TIter forwards the call to the TIterator embedded in the TObjArrayIter
3. TIterator forwards the call to the TObjArrayIter
4. TObjArrayIter finds the next MyClass object and returns it
5. TIter passes the MyClass object back to the caller

Sometimes the TIter object is called next, and then instead of writing:
next.Next()
Which is legal, but looks rather odd, iteration is written as:
next()
This works because the function operator() is defined for the TIter class to be equivalent to the Next() method.

---

### The TList Collection

A TList is a doubly linked list. Before being inserted into the list the object pointer is wrapped in a TObjLink object that contains, besides the object pointer also a previous and next pointer.

Objects are typically added using:

- Add()
- AddFirst(), AddLast()
- AddBefore(), AddAfter()

**Main features of TList:** very low cost of adding/removing elements anywhere in the list.

**Overhead per element:** 1 TObjLink, i.e. two 4 (or 8) byte pointers + pointer to vtable = 12 (or 24) bytes.

```c
class TList : public TSeqCollection
{
private:
    TObjLink *fLast;
    TObjLink *fFirst;
    . . .
};
```

```c
class TObjLink {
friend class TList;
private:
    TObjLink *fPrev;
    TObjLink *fNext;
    TObject   *fObject;
    . . .
};
```

The diagram below shows the internal data structure of a TList:
Iterating over a TList

There are basically four ways to iterate over a TList:

1. Using the ForEach script:
   ```
   GetListOfPrimitives() -> ForEach(TObject, Draw());
   ```

2. Using the TList iterator TListIter (via the wrapper class TIter):
   ```
   TIter next(GetListOfTracks());
   while ((TTrack *obj = (TTrack *) next()))
      obj->Draw();
   ```

3. Using the TObjectLink list entries (that wrap the TObject *):
   ```
   TObjectLink *lnk = GetListOfPrimitives() -> FirstLink();
   while (lnk) {
      lnk->GetObject()->Draw();
      lnk = lnk->Next();
   }
   ```

4. Using the TList's After() and Before() member functions:
   ```
   TFree *idcur = this;
   while (idcur) {
      ... idcur = (TFree*) GetListOfFree() -> After(idcur);
   }
   ```

Method 1 uses internally method 2.
Method 2 works for all collection classes. TIter overloads operator().
Methods 3 and 4 are specific for TList.
Methods 2, 3 and 4 can also easily iterate backwards using either a backward TIter (using argument kIterBackward) or by using LastLink() and lnk->Prev() or by using the Before() method.

The TObjectCollection

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

At creation time one specifies the default array size (default = 16) and lower bound (default = 0). Resizing involves a re-allocation and a copy of the old array to the new. This can be costly if done too often. If possible, set initial size close to expected final size. Index validity is always checked (if you are 100% sure and maximum performance is needed you can use UnCheckedAt() instead of At() or operator[]).

If the stored objects are sortable the array can be sorted using Sort(). Once sorted, efficient searching is possible via the BinarySearch() method.

Iterating can be done using a TIter iterator or via a simple for loop:

```
for (int i = 0; i <= fArr.GetLast(); i++)
   if ((track = (TTrack*)fArr[i])) // or fArr.At(i)
      track->Draw();
```
**TClonesArray – An Array of Identical Objects**

A TClonesArray is an array of identical (clone) objects. The memory for the objects stored in the array is allocated only once in the lifetime of the clones array. All objects must be of the same class and the object must have a fixed size (i.e. they may not allocate other objects). For the rest this class has the same properties as a TObjArray.

The class is specially designed for repetitive data analysis tasks, where in a loop many times the same objects are created and deleted.

The diagram below shows the internal data structure of a TClonesArray:

```cpp
class TClonesArray : public TObjArray {
private:
  TObjArray *fKeep;
  TClass       *fClass;
  ...
};
```

The Idea Behind TClonesArray

To reduce the very large number of new and delete calls in large loops like this (O(100000) x O(10000) times new/delete):

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

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

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

Considering that a pair of new/delete calls on average cost about 70 µs, O(10^9) new/deletes will save about 19 hours.

For the other collections see the class reference guide on the web and the test program $ROOTSYS/test/tcollex.cxx.

**Template Containers and STL**

Some people dislike polymorphic containers because they are not truly “type safe”. In the end, the compiler leaves it to the user to ensure that the types are correct. This only leaves the other alternative: creating a new class each time a new (container organization) / (contained object) combination is needed. To say the least this could be very tedious. Most people faced with this choice would, for each type of container:

1. Define the class leaving a dummy name for the contained object type.
2. When a particular container was needed, copy the code and then do a global search and replace for the contained class.

C++ has a built in template scheme that effectively does just this. For example:

```cpp
template<class T>
class ArrayContainer {
private:
  T *member[10];
  ...
};
```

This is an array container with a 10-element array of pointers to T, it could hold up to 10 T objects. This array is flawed because it is static and hard-coded, it should be dynamic. However, the important point is that the template statement indicates that T is a template, or parameterized class. If we need an ArrayContainer for Track objects, it can be created by:

```cpp
ArrayContainer<Track> MyTrackArrayContainer;
```

C++ takes the parameter list, and substitutes Track for T throughout the definition of the class ArrayContainer, then compiles the code so generated, effectively doing the same we could do by hand, but with a lot less effort. This produces code that is type safe, but does have different drawbacks:

- Templates make code harder to read.
• At the time of writing this documentation, some compilers can be very slow when dealing with templates.
• It does not solve the problem when a container has to hold a heterogeneous set of objects.
• The system can end up generating a great deal of code; each container/object combination has its own code, a phenomenon that is sometimes referred to as code bloat.

The Standard Template Library (STL) is part on ANSI C++, and includes a set of template containers.
The physics vector classes describe vectors in three and four dimensions and their rotation algorithms. The classes were ported to root from CLHEP see: http://wwwinfo.cern.ch/asd/lhc++/clhep/manual/UserGuide/Vector/vector.html

The Physics Vector Classes

In order to use the physics vector classes you will have to load the Physics library:

```
gSystem.Load("libPhysics.so");
```

There are four classes in this package. They are:

- **TVector3**: A general three-vector. A TVector3 may be expressed in Cartesian, polar, or cylindrical coordinates. Methods include dot and cross products, unit vectors and magnitudes, angles between vectors, and rotations and boosts. There are also functions of particular use to HEP, like pseudo-rapidity, projections, and transverse part of a TVector3, and kinetic methods on 4-vectors such as Invariant Mass of pairs or containers of particles.

- **TLorenzVector**: A general four-vector class, which can be used either for the description of position and time \((x, y, z, t)\) or momentum and energy \((p_x, p_y, p_z, E)\).

- **TRotation**: a class describing a rotation of a TVector3 object.

- **TLorenzRotation**: a class to describe the Lorentz transformations including Lorentz boosts and rotations.

There is also a TVector2, it is a basic implementation of a vector in two dimensions and not part of the CLHEP translation.

TVector3

TVector3 is a general three vector class, which can be used for description of different vectors in 3D. Components of three vector:

- \(x, y, z\) - basic components
- \(\theta\) = azimuth angle
- \(\phi\) = polar angle
- magnitude = \(mag = \sqrt{x^2 + y^2 + z^2}\)
- transverse component = \(perp = \sqrt{x^2 + y^2}\)

Using the TVector3 class you should remember that it contains only common features of three vectors and lacks methods specific for some particular vector values. For example, it has no translate function because translation has no meaning for vectors.

Declaration / Access to the components

TVector3 has been implemented as a vector of three Double_t variables, representing the Cartesian coordinates. By default the values are initialized to zero, however you can change them in the constructor:

```
TVector3 v1;   // v1 = (0,0,0)
TVector3 v2(1); // v2 = (1,0,0)
TVector3 v3(1,2,3); // v3 = (1,2,3)
TVector3 v4(v2); // v4 = v2
```

It is also possible (but not recommended) to initialize a TVector3 with a Double_t or Float_t C array.

You can get the components by name or by index:

```
xx = v1.X();   or   xx = v1(0);
yy = v1.Y();        yy = v1(1);
zz = v1.Z();      zz = v1(2);
```
The methods \texttt{SetX()}, \texttt{SetY()}, \texttt{SetZ()} and \texttt{SetXYZ()} allows you to set the components:

\begin{verbatim}
v1.SetX(1.); v1.SetY(2.); v1.SetZ(3.);
v1.SetXYZ(1.,2.,3.);
\end{verbatim}

Other Coordinates

To get information on the \texttt{TVector3} in spherical (\rho, \phi, \theta) or cylindrical (\rho, \phi, \zeta) coordinates, the following methods can be used.

\begin{verbatim}
Double_t m = v.Mag();  // get magnitude (=\rho=\sqrt{x^2+y^2+z^2})
Double_t m2 = v.Mag2(); // get magnitude squared
Double_t t = v.Theta(); // get polar angle
Double_t ct = v.CosTheta(); // get cos of theta
Double_t p = v.Phi(); // get azimuth angle
Double_t pp = v.Perp(); // get transverse component
Double_t pp2 = v.Perp2(); // get transverse squared
\end{verbatim}

It is also possible to get the transverse component with respect to another vector:

\begin{verbatim}
Double_t ppv1 = v.Perp(v1);
Double_t ppv2 = v.Perp2(v1);
\end{verbatim}

The pseudo-rapidity \((\eta = -\ln \tan(\phi/2))\) can be get by \texttt{Eta()} or \texttt{PseudoRapidity()}:

\begin{verbatim}
Double_t eta = v.PseudoRapidity();
\end{verbatim}

These setters change one of the non-Cartesian coordinates:

\begin{verbatim}
v.SetTheta(.5); // keeping \rho and \phi
v.SetPhi(.8);  // keeping \rho and \theta
v.SetMag(10.); // keeping \theta and \phi
v.SetPerp(3.); // keeping \zeta and \phi
\end{verbatim}

Arithmetic / Comparison

The \texttt{TVector3} class has operators to add, subtract, scale and compare vectors:

\begin{verbatim}
v3 = -v1;
v1 = v2+v3;
v1 += v3;
v1 -= v3;
v1 = v1 * v3;
v1 = v1 / v3;
if(v1 == v2) {...}
if(v1 != v2) {...}
\end{verbatim}

Related Vectors

\begin{verbatim}
v2 = v1.Unit();    // get unit vector parallel to v1
v2 = v1.Orthogonal(); // get vector orthogonal to v1
\end{verbatim}

Scalar and Vector Products

\begin{verbatim}
s = v1.Dot(v2);  // scalar product
s = v1 * v2;    // scalar product
v = v1.Cross(v2); // vector product
\end{verbatim}

Angle between Two Vectors

\begin{verbatim}
Double_t a = v1.Angle(v2);
\end{verbatim}

Rotation around Axes

\begin{verbatim}
v.RotateX(.5);  // rotation around \phi
v.RotateY(TMath::Pi()); // rotation around \theta
v.RotateZ(angle); // rotation around \rho
\end{verbatim}

Rotation around a Vector

\begin{verbatim}
v1.Rotate(TMath::Pi()/4, v2); // rotation around v2
\end{verbatim}

Rotation by TRotation

\texttt{TVector3} objects can be rotated by \texttt{TRotation} objects using the \texttt{Transform()} method, the operator \texttt{*=}, or the operator \texttt{*} of the \texttt{TRotation} class. See the later section on \texttt{TRotation}.

\begin{verbatim}
TRotation m;
... v1.transform(m);
v1 *= m; // v1 = m*v1
\end{verbatim}

Transformation from Rotated Frame

This code transforms \texttt{v1} from the rotated frame (\zeta' parallel to direction, \x' in the theta plane and \y' in the xy plane as well as perpendicular to the theta plane) to the (x, y, z) frame.

\begin{verbatim}
v1.RotateUz(direction);
// direction must be TVector3 of unit length
\end{verbatim}
The TRotation class describes a rotation of TVector3 object. It is a 3 * 3 matrix of Double_t:

\[
\begin{pmatrix}
xx & xy & xz \\
yx & yy & yz \\
zx & zy & zz \\
\end{pmatrix}
\]

It describes a so-called active rotation, i.e. a rotation of objects inside a static system of coordinates. In case you want to rotate the frame and want to know the coordinates of objects in the rotated system, you should apply the inverse rotation to the objects. If you want to transform coordinates from the rotated frame to the original frame you have to apply the direct transformation.

A rotation around a specified axis means counterclockwise rotation around the positive direction of the axis.

**Declaration, Access, Comparisons**

```cpp
TRotation r;  // r initialized as identity
TRotation m(r);  // m = r
```

There is no direct way to set the matrix elements - to ensure that a TRotation always describes a real rotation. But you can get the values by with the methods XX()..ZZ() or the (,) operator:

```cpp
Double_t xx = r.XX();  // the same as xx=r(0,0)
xx = r(0,0);
if (r==m) {...}  // test for equality
if (r!=m) {..}  // test for inequality
if (r.IsIdentity()) {...}  // test for identity
```

**Rotation Around Axes**

The following matrices describe counter-clockwise rotations around the coordinate axes and are implemented in RotateX(), RotateY() and RotateZ():

\[
Rx[a] = \begin{pmatrix}
1 & 0 & 0 \\
0 & \cos(a) & -\sin(a) \\
0 & \sin(a) & \cos(a) \\
\end{pmatrix}
\]

\[
Ry[a] = \begin{pmatrix}
\cos(a) & 0 & \sin(a) \\
0 & 1 & 0 \\
-\sin(a) & 0 & \cos(a) \\
\end{pmatrix}
\]

\[
Rz[a] = \begin{pmatrix}
\cos(a) & -\sin(a) & 0 \\
\sin(a) & \cos(a) & 0 \\
0 & 0 & 1 \\
\end{pmatrix}
\]

**Rotation around Arbitrary Axis**

The Rotate() method allows you to rotate around an arbitrary vector (not necessary a unit one) and returns the result.

```cpp
r.Rotate(TMath::Pi()/3,TVector3(3,4,5));
```

It is possible to find a unit vector and an angle, which describe the same rotation as the current one:

```cpp
Double_t angle;  
TVector3 axis;  
r.GetAngleAxis(angle,axis);
```

**Rotation of Local Axes**

The RotateAxes() method adds a rotation of local axes to the current rotation and returns the result:

```cpp
TVector3 newX(0,1,0);  
TVector3 newY(0,0,1);  
TVector3 newZ(1,0,0);  
a.RotateAxes(newX,newY,newZ);  
```

**Methods** ThetaX(), ThetaY(), ThetaZ(), PhiX(), PhiY(), PhiZ() return azimuth and polar angles of the rotated axes:

```cpp
Double_t tx,ty,tz,px,py,pz;  
tx= a.ThetaX();  
...
px= a.PhiX();
```

**Inverse Rotation**

```cpp
TRotation a,b;  
...

b = a.Inverse();  // b is inverse of a, a is unchanged
b = a.Invert();  // invert a and set b = a
```

**Compound Rotations**

The operator * has been implemented in a way that follows the mathematical notation of a product of the two matrices which describe the two consecutive rotations. Therefore the second rotation should be placed first:

```cpp
r.RotateX(TMath::Pi());  // rotation around the x-axis
```
Rotation of TVector3
The TRotation class provides an operator * which allows to express a rotation of a TVector3 analog to the mathematical notation
\[
\begin{array}{c|ccc}
  x' & xx & xy & xz \\
  y' & yx & yy & yz \\
  z' & zx & zy & zz \\
\end{array}
\]

You can also use the Transform() method or the operator *= of the TVector3 class:

```cpp
TVector3 v(1,1,1);
TRotation r;
v = r * v;
```

TLorentzVector
TLorentzVector is a general four-vector class, which can be used either for the description of position and time \((x, y, z, t)\) or momentum and energy \((p_x, p_y, p_z, E)\).

Declaration
TLorentzVector has been implemented as a set a TVector3 and a Double_t variable. By default all components are initialized by zero.

```cpp
TLorentzVector v1; // initialized by (0., 0., 0., 0.)
TLorentzVector v2(1., 1., 1., 1.);
TLorentzVector v3(v1);
TLorentzVector v4(TVector3(1., 2., 3.),4.);
```

Access to Components
There are two sets of access functions to the components of a TLorentzVector: \(X(), Y(), Z(), T()\) and \(Px(), Py(), Pz()\) and \(E()\). Both sets return the same values but the first set is more relevant for use where TLorentzVector describes a combination of position and time and the second set is more relevant where TLorentzVector describes momentum and energy:

```cpp
Double_t xx = v.X();
...
Double_t tt = v.T();
Double_t px = v.Px();
...
Double_t ee = v.E();
```

The components of TLorentzVector can also accessed by index:

```cpp
xx = v(0);       or     xx = v[0];
yy = v(1);              yy = v[1];
zz = v(2);              zz = v[2];
tt = v(3);              tt = v[3];
```

You can use the Vect() method to get the vector component of TLorentzVector:

```cpp
TVector3 p = v.Vect();
```
For setting components there are two methods: SetX(),...
SetPx(),...

v.SetX(1.); or v.SetPx(1.);
... ...
v.SetT(1.); v.SetE(1.);

To set more the one component by one call you can use the SetVect() function for the TVector3 part or SetXY() for the TVector2 part. For convenience there is also a SetXYZM() function:

v.SetVect(TVector3(1,2,3));
v.SetXYZT(x,y,z,t);
... ...
v.SetPxPyPzE(px,py,pz,e);
... ...
// V=(x,y,z,e=Sqrt(x*x+y*y+z*z+m*m))

Vector Components in non-Cartesian Coordinates

There are a couple of methods to get and set the TVector3 part of the parameters in spherical coordinate systems:

Double_t m, theta, cost, phi, pp, pp2, ppv2, pp2v2;
m = v.Rho();
t = v.Theta();
... ...
cost = v.CosTheta();
phi = v.Phi();
v.SetRho(10.);
v.SetTheta(TMath::Pi()*.3);
v.SetPhi(TMath::Pi());
... ...
// set the components
v.SetXYZM(x,y,z,m);

or get information about the r-coordinate in cylindrical systems:

Double_t pp, pp2, ppv2, pp2v2;
... ...
pp = v.Perp(); // get transverse component
pp2 = v.Perp2(); // get transverse component squared
ppv2 = v.Perp(v1); // get transvers component with respect to another vector
pp2v2 = v.Perp(v1);

or for convenience there are two more set functions
SetPtEtaPhiIE(pt,eta,phi,e); and
SetPtEtaPhiIM(pt,eta,phi,m);

Arithmetic and Comparison Operators

The TLorentzVector class provides operators to add, subtract or compare four-vectors:

v3 = -v1;
v1 = v2+v3;
v1+= v3;
v1 = v2 + v3;
v1-= v3;
if (v1 == v2) {...}
if(v1 != v3) {...}

Magnitude/Invariant mass, beta, gamma, scalar product

The scalar product of two four-vectors is calculated with the (-,-,-,+) metric:

s = v1*v2 = t1*t2-x1*x2-y1*y2-z1*z2

The magnitude squared mag2 of a four-vector is therefore:

mag2 = v*Mag2(); or mag2 = v.M2();

Since in case of momentum and energy the magnitude has the meaning of invariant mass TLorentzVector provides the more meaningful aliases M2() and M();

The methods Beta() and Gamma() returns beta and gamma = 1/Sqrt(1-beta*beta).

Lorentz Boost

A boost in a general direction can be parameterized with three parameters which can be taken as the components of a three vector b = (bx,by,bz).
With x = (x,y,z) and gamma = 1/Sqrt(1-beta*beta), an arbitrary active Lorentz boost transformation (from the rod frame to the original frame) can be written as:

x' = x' + (gamma-1)/(beta*beta)*(b*x') * b + gamma * t' * b = gamma *(t' + b*x').
The Boost() method performs a boost transformation from the rod frame to the original frame. BoostVector() returns a TVector3 of the spatial components divided by the time component:

```cpp
TVector3 b;  
v.Boost(bx,by,bz);  
v.Boost(b);  
b = v.BoostVector();  // b=(x/t,y/t,z/t)
```

### Rotations
There are four sets of functions to rotate the TVector3 component of a TLorentzVector:

**Rotation around Axes**

```cpp
v.RotateX(TMath::Pi()/2.);  
v.RotateY(.5);  
v.RotateZ(.99);  
```

**Rotation around an Arbitrary Axis**

```cpp
v.Rotate(TMath::Pi()/4., v1);  // rotation around v1
```

**Transformation from Rotated Frame**

```cpp
v.RotateUz(direction);  // direction must be a unit TVector3
```

### Transformation by TLorentzRotation
A general Lorentz transformation see class TLorentzRotation can be used by the Transform() method, the * or * operator of the TLorentzRotation class:

```cpp
TLorentzRotation l;  
v.Transform(l);  
v = l*v;  or  v *= l;  // v = l*v
```

### TLorentzRotation
The TLorentzRotation class describes Lorentz transformations including Lorentz boosts and rotations (see TRotation):

\[
\begin{pmatrix}
xx & xy & xz & xt \\
yx & yy & yz & yt \\
zx & zy & zz & zt \\
tx & ty & tz & tt
\end{pmatrix}
\]

**Declaration**

By default it is initialized to the identity matrix, but it may also be initialized by an other TLorentzRotation, by a pure TRotation or by a boost:

```cpp
TLorentzRotation l;  // l is initialized as identity  
TLorentzRotation m(l);  // m = l  
TRotation r;  
TLorentzRotation lr(r);  
TLorentzRotation lb1(bx,by,bz);  
TLorentzRotation lb2(b);
```

The Matrix for a Lorentz boosts is:

\[
\begin{pmatrix}
1+\gamma'^2 & \gamma'bx & \gamma'by & \gamma'bz \\
\gamma'bx & 1+\gamma'^2 & \gamma'by & \gamma'bz \\
\gamma'by & \gamma'by & 1+\gamma'^2 & \gamma'bz \\
\gamma'bx & \gamma'by & \gamma'bz & 1+\gamma'^2
\end{pmatrix}
\]

with the boost vector \(b=(bx,by,bz)\) and \(\gamma = 1/\sqrt{1-\beta^2}\) and \(\gamma' = (\gamma-1)/\beta^2\).

### Miscellaneous

**Angle Between Two Vectors**

```cpp
Double_t a = v1.Angle(v2);  // get angle between v1 and v2
```

**Light-cone Components**

Methods Plus() and Minus() return the positive and negative light-cone components:

```cpp
Double_t pcone = v.Plus();  
Double_t mcone = v.Minus();
```
Access to the matrix Components/Comparisons

Access to the matrix components is possible with the methods XX(), XY() .. TT(), and with the operator (int,int):

```cpp
Double_t xx;
TLorentzRotation l;
xx = l.XX(); // gets the xx component
xx = l[0,0]; // gets the xx component
if (l==m) {...} // test for equality
if (l.IsIdentity()) {...} // test for identity
```

Transformations of a Lorentz Rotation

Compound transformations

There are four possibilities to find the product of two TLorentzRotation transformations:

```cpp
TLorentzRotation a,b,c;
c = b*a; // product
a *= b; // a is unchanged
c = a.Transform(b); // a*b*a then c=a
```

Lorentz boosts

```cpp
Double_t bx, by, bz;
TVector3 v(bx,by,bz);
TLorentzRotation l;
l.Boost(v);
l.Boost(bx,by,bz);
```

Rotations

```cpp
TVector3 axis;
l.RotateX(TMath::Pi()); // rotation around x-axis
l.Rotate(.5,axis); // rotation around specified vector
```

Inverse transformation

The matrix for the inverse transformation of a TLorentzRotation is as follows:

```
| xx  yx  zx -tx |
| xy  yy  zy -ty |
| xz  yz  zz -tz |
| -xt -yt -zt  tt |
```

To return the inverse transformation keeping the current one unchanged, use the method Inverse(). Invert() inverts the current TLorentzRotation:

```cpp
l1 = l2.Inverse(); // l1 is inverse of l2, l2 unchanged
l1 = l2.Invert(); // invert l2, then l1=l2
```

Transformation of a TLorentzVector

To apply TLorentzRotation to TLorentzVector you can use either the VectorMultiplication() method or the * operator. You can also use the Transform() function and the *= operator of the TLorentzRotation class.

```cpp
TLorentzVector v;
TLorentzRotation l;
...
v=1.VectorMultiplication(v);
v = l * v;
v.Transform(l);
v *= l; // v = l*v
```

Physics Vector Example

To see an example of using physics vectors you can look at the test file. It is in $ROOTSYS/test/TestVectors.cxx. The vector classes are not loaded by default, and to run it, you will need to load libPhysics.so first:

```
root [] .L $ROOTSYS/lib/libPhysics.so
root [] .x TestVectors.cxx
```

To load the physics vector library in a ROOT application use:

```
gSystem->Load("libPhysics");
```

The example $ROOTSYS/test/TestVectors.cxx does not return much, especially if all went well, but when you look at the code you will find examples for many calls.
The Tutorials and Tests

This chapter is a guide to the examples that come with the installation of ROOT. They are located in two directories:

$ROOTSYS/tutorials

and

$ROOTSYS/test

The tutorials directory contains many example scripts. For the examples to work you must have write permission and you will need to execute hsimple.C first. If you do not have write permission in the $ROOTSYS/tutorials directory, copy the entire directory to your area.

The script hsimple.C displays a histogram as it is being filled, and creates a ROOT file used by the other examples. To execute it type:

```
root [] .x hsimple.C
```

Now execute demos.C which brings up the button bar shown on the left. You can click on any button to execute an other example. To see the source, open the corresponding source file (for example fit1.C). Once you are done, and want to quit the ROOT session, you can do so by typing .q.

```
root [] .q
```

$ROOTSYS/test

The test directory contains a set of examples that represent all areas of the framework. When a new release is cut, the examples in this directory are compiled and run to test the new release's backward compatibility.

We see these source files:

- hsimple.cxx - Simple test program that creates and saves some histograms
- MainEvent.cxx - Simple test program that creates a ROOT Tree object and fills it with some simple structures but also with complete histograms. This program uses the files Event.cxx, EventCint.cxx and Event.h. An example of a procedure to link this program is in bind_Event. Note that the Makefile invokes the rootcint utility to generate the CINT interface EventCint.cxx.
- Event.cxx - Implementation for classes Event and Track
- minexam.cxx - Simple test program to test data fitting.
- tcollex.cxx - Example usage of the ROOT collection classes.
- tcollbm.cxx - Benchmarks of ROOT collection classes
- tstring.cxx - Example usage of the ROOT string class.
- Tmatrix.cxx - Verification program for the TMatrix class.
- tvector.cxx - Verification program for the TVector class.
- vlazy.cxx - Verification program for lazy matrices.
- hwfords.cxx - Small program showing basic graphics.
- guittest.cxx - Example usage of the ROOT GUI classes.
- Hello.cxx - Dancing text example
- Aclock.cxx - Analog clock (a la X11 xclock)
- Tetris.cxx - The famous Tetris game (using ROOT basic graphics).
- stress.cxx - Important ROOT stress testing program.

The $ROOTSYS/test directory is a gold mine of root-wisdom nuggets, and we encourage you to explore and exploit it. These instructions will compile all programs in $ROOTSYS/test:

1. If you do not have write permission in the $ROOTSYS/test directory, copy the entire $ROOTSYS/test directory to your area.

```
root [] .x demos.C
```
2. The Makefile is a useful example of how ROOT applications are linked and built. Edit the Makefile to specify your architecture by changing the ARCH variable, for example, on an SGI machine type:

```
ARCH = sgikcc
```

3. Now compile all programs:

```
% gmake
```

This will build several applications and shared libraries. We are especially interested in Event, stress, and guitest.

**Event – An Example of a ROOT Application**

Event is created by compiling MainEvent.cxx and Event.cxx. It creates a ROOT file with a tree and two histograms.

When running Event we have four optional arguments with defaults:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number of Events (1 ... n)</td>
</tr>
<tr>
<td>2</td>
<td>Compression level:</td>
</tr>
<tr>
<td>0</td>
<td>no compression at all.</td>
</tr>
<tr>
<td>1</td>
<td>If the split level is set to zero, everything is compressed according to the gzip level 1. If split level is set to 1, leaves that are not floating point numbers are compressed using the gzip level 1.</td>
</tr>
<tr>
<td>2</td>
<td>If the split level is set to zero, everything is compressed according to the gzip level 2. If split level is set to 1, all non floating point leaves are compressed according to the gzip level 2 and the floating point leaves are compressed according to the gzip level 1 (gzip level -1).</td>
</tr>
<tr>
<td>3</td>
<td>Split or not Split</td>
</tr>
<tr>
<td>0</td>
<td>only one single branch is created and the complete event is serialized in one single buffer</td>
</tr>
<tr>
<td>1</td>
<td>a branch per variable is created.</td>
</tr>
<tr>
<td>4</td>
<td>Fill</td>
</tr>
<tr>
<td>0</td>
<td>read the file</td>
</tr>
<tr>
<td>1</td>
<td>write the file, don’t fill the histograms</td>
</tr>
<tr>
<td>2</td>
<td>write, don’t fill the histograms</td>
</tr>
<tr>
<td>10</td>
<td>fill the histograms, don’t write the file</td>
</tr>
<tr>
<td>11</td>
<td>fill the histograms, write the file</td>
</tr>
<tr>
<td>20</td>
<td>read the file sequentially</td>
</tr>
<tr>
<td>25</td>
<td>read the file at random</td>
</tr>
</tbody>
</table>

**Effect of Compression on File Size and Write Times**

Floating point numbers are compressed differently because the gain when compressing them is about 20-30%. For other data types it is generally better and around 100%.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Split (Split)</td>
</tr>
<tr>
<td>0</td>
<td>(Write, no fill)</td>
</tr>
</tbody>
</table>

**Not competitive. They take up to much write time compared to the gain in file space.**

Below are three runs of Event on a Pentium III 650 Mhz and the resulting file size and write and read times.

**No Compression:**

```
> Event 400 0 1 1
400 events and 19153182 bytes processed.
RealTime=6.840000 seconds, CpuTime=3.560000 seconds
compression level=0, split=1, arg4=1
You write 2.800173 Mbytes/Realtime seconds
You write 5.380107 Mbytes/CpuTime seconds
```

```
> ls -l Event.root
-rw-r--r-- 1 19752171 Feb 23 18:26 Event.root
```

```
> Event 400 1 1 1
400 events and 19153182 bytes processed.
RealTime=6.440000 seconds, CpuTime=4.020000 seconds
compression level=1, split=1, arg4=1
You write 2.974096 Mbytes/Realtime seconds
You write 4.764473 Mbytes/CpuTime seconds
```

```
> ls -l Event.root
-rw-r--r-- 1 17728188 Feb 23 18:28 Event.root
```

```
> Event 400 1 1 20
400 events and 19153182 bytes processed.
RealTime=0.790000 seconds, CpuTime=0.790000 seconds
You read 24.244533 Mbytes/Realtime seconds
You read 24.244533 Mbytes/CpuTime seconds
```

We see the file size without compression is 19.75 MB, the write time is 6.84 seconds and the read time is 0.79 seconds.

**Compression = 1: event is compressed:**

```
> Event 400 1 1 1
400 events and 19153182 bytes processed.
RealTime=6.440000 seconds, CpuTime=4.020000 seconds
compression level=1, split=1, arg4=1
You write 2.974096 Mbytes/Realtime seconds
You write 4.764473 Mbytes/CpuTime seconds
```

```
> ls -l Event.root
-rw-r--r-- 1 17728188 Feb 23 18:28 Event.root
```

```
> Event 400 1 1 20
400 events and 19153182 bytes processed.
RealTime=0.900000 seconds, CpuTime=0.900000 seconds
You read 21.281312 Mbytes/Realtime seconds
You read 21.281312 Mbytes/CpuTime seconds
```

We see the file size 17.73, the write time was 6.44 seconds and the read time was 0.9 seconds.
Compression = 2: Floating point numbers are compressed with level 1:

```bash
$ Event 400 2 1 1
```

400 events and 19153182 bytes processed.
RealTime=11.340000 seconds, CpuTime=9.510000 seconds
compression level=2, split=1, arg4=1
You write 1.688993 Mbytes/Realtime seconds
You write 2.014004 Mbytes/CpuTime seconds

```bash
$ ls -l Event.root
```

…     13783799 Feb 23 18:29 Event.root

```bash
$ Event 400 2 1 20
```

400 events and 19153182 bytes processed.
RealTime=2.170000 seconds, CpuTime=2.170000 seconds
You read 8.826351 Mbytes/Realtime seconds
You read 8.826351 Mbytes/CpuTime seconds

The file size is 13.78 MB, the write time is 11.34 seconds and the read time is 2.17 seconds.

This table summarizes the findings on the impact of compressions:

<table>
<thead>
<tr>
<th>Compression</th>
<th>File Size</th>
<th>Write Times</th>
<th>Read Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>19.75 MB</td>
<td>6.84 sec.</td>
<td>0.79 sec.</td>
</tr>
<tr>
<td>1</td>
<td>17.73 MB</td>
<td>6.44 sec.</td>
<td>0.90 sec.</td>
</tr>
<tr>
<td>2</td>
<td>13.78 MB</td>
<td>11.34 sec.</td>
<td>2.17 sec.</td>
</tr>
</tbody>
</table>

### Setting the Split Level

**Split Level = 0:**

Now we execute Event with the split parameter set to 0:

```bash
$ Event 400 1 0 1
```

We notice that only one branch is visible (event). The individual data members of the Event object are no longer visible in the browser. They are contained in the event object on the event branch, because we specified no splitting.

**Split Level = 1:**

Setting the split level to 1 will create a branch for each data member in the Event object. We can see this by browsing the resulting files.

First we execute Event and set the split level to 1 and start the browser to examine the split tree:

```bash
$ Event 400 1 1 1
```

The executable **stress** is created by compiling **stress.cxx**. It completes sixteen tests covering the following capabilities of the ROOT framework.

1. Functions, Random Numbers, Histogram Fits
2. Size & compression factor of a ROOT file
3. Purge, Reuse of gaps in TFile
4. 2D Histograms, Functions, 2D Fits
5. Graphics & PostScript
6. Subdirectories in a ROOT file
7. TNtuple, Selections, TCut, TCutG, TEventList
8. Split and Compression modes for Trees
9. Analyze Event.root file of stress 8
10. Create 10 files starting from Event.root
11. Test chains of Trees using the 10 files
12. Compare histograms of test 9 and 11
13. Merging files of a chain
14. Check correct rebuild of Event.root in test 13
15. Divert Tree branches to separate files
16. CINT test (3 nested loops) with LHCb trigger

The program **stress** takes one argument, the number of events to process. The default is 1000 events. Be aware that executing stress with 1000 events will create several files consuming about 100 MB of disk space.
running stress with 30 events will consume about 20 MB. The disk space is released once stress is done.

There are two ways to run stress:

From the system prompt or from the ROOT prompt using the interpreter. Start ROOT with the batch mode option (-b) to suppress the graphic output.

```
> cd $ROOTSYS/test
> stress // default 1000 events
> stress 30 // test with 30 events
```

The output of stress includes a pass/fail conclusion for each test, the total number of bytes read and written, and the elapsed real and CPU time. It also calculates a performance index for your machine relative to a reference machine a DELL Inspiron 7500 (Pentium III 600 MHz) with 256 MB of memory and 18 GBytes IDE disk in ROOTMARKS. Higher ROOTMARKS means better performance. The reference machine has 200 ROOTMARKS, so the sample run below with 53.7 ROOTMARKS is about four times slower than the reference machine.

Here is a sample run:

```
% root -b
root [] .x stress.cxx (30)
```

```
Test 1 : Functions, Random Numbers, Histogram Fits............. OK
Test 2 : Check size & compression factor of a Root file........ OK
Test 3 : Purge, Reuse of gaps in TFile......................... OK
Test 4 : Test of 2-d histograms, functions, 2-d fits........... OK
Test 5 : Test graphics & PostScript............................OK
Test 6 : Test subdirectories in a Root file.................... OK
Test 7 : TNtuple, selections, Tcut, Tcuts, TEventlist........ OK
Test 8 : Trees split and compression modes..................... OK
Test 9 : Analyze Event.root file of stress 8................... OK
Test 10 : Create 10 files starting from Event.root............ OK
Test 11 : Test chains of Trees using the 10 files............. OK
Test 12 : Compare histograms of test 9 and 11................... OK
Test 13 : Test merging files of a chain......................... OK
Test 14 : Check correct rebuilt of Event.root in test 13..... OK
Test 15 : Divert Tree branches to separate files.............. OK
Test 16 : CINT test (3 nested loops) with LHCb trigger....... OK
```

```
stress    : Total I/O =   75.3 Mbytes, I =   59.2, O =  16.1
stress    : Compr I/O =   75.7 Mbytes, I =   60.0, O =  15.7
stress    : Real Time = 307.61 seconds Cpu Time = 292.82 seconds
```

```
* ROOTMARKS = 53.7 * Root2.25/00   20000710/1022
```

The guitest example, created by compiling guitest.cxx, tests and illustrates the use of the native GUI widgets such as cascading menus, dialog boxes, sliders and tab panels. It is a very useful example to study when designing a GUI. Below are some examples of the output of guitest, to run it type guitest at the system prompt in the $ROOTSYS/test directory.

We have included an entire chapter on this subject where we explore guitest in detail and use it to explain how to build our own ROOT application with a GUI (see Chapter Writing a Graphical User Interface).
17 Example Analysis

This chapter is an example of a typical physics analysis. Large data files are chained together and analyzed using the TSelector class.

Explanation

This script uses four large data sets from the H1 collaboration at DESY Hamburg. One can access these data sets (277 Mbytes) from the ROOT web site at: ftp://root.cern.ch/root/h1analysis/
The physics plots generated by this example cannot be produced using smaller data sets.

There are several ways to analyze data stored in a ROOT Tree

- **Using TTree::Draw:**
  This is very convenient and efficient for small tasks. A TTree::Draw call produces one histogram at the time. The histogram is automatically generated. The selection expression may be specified in the command line.
  - Using the TTreeViewer:
    This is a graphical interface to TTree::Draw with the same functionality.
- **Using the code generated by TTree::MakeClass:**
  In this case, the user creates an instance of the analysis class. He has the control over the event loop and he can generate an unlimited number of histograms.
- **Using the code generated by TTree::MakeSelector:**
  Like for the code generated by TTree::MakeClass, the user can do complex analysis. However, he cannot control the event loop. The event loop is controlled by TTree::Process called by the user. This solution is illustrated by the code below. The advantage of this method is that it can be run in a parallel environment using PROOF (the Parallel Root Facility).

A chain of four files (originally converted from PAW ntuples) is used to illustrate the various ways to loop on ROOT data sets. Each contains a ROOT Tree named "h42". The class definition in h1analysis.h has been generated automatically by the ROOT utility TTree::MakeSelector using one of the files with the following statement:

```
TTree::MakeSelector("h1analysis");
```

This produces two files: h1analysis.h and h1analysis.C. A skeleton of h1analysis.C file is made for you to customize. The h1analysis class is derived from the ROOT class TSelector. The following members functions of h1analysis (i.e. TSelector) are called by the TTree::Process method.

- **Begin:** This function is called every time a loop over the tree starts. This is a convenient place to create your histograms.
- **Notify():** This function is called at the first entry of a new tree in a chain.
- **ProcessCut:** This function is called at the beginning of each entry to return a flag true if the entry must be analyzed.
- **ProcessFill:** This function is called in the entry loop for all entries accepted by Select.
- **Terminate:** This function is called at the end of a loop on a TTree.

This is a convenient place to draw and fit your histograms.

To use this program, try the following session.
First, turn the timer on to show the real and CPU time per command.

```
root[] gROOT->Time();
```

**Step A:** create a TChain with the four H1 data files. The chain can be created by executing this short script h1chain.C below. $H1$ is a system symbol pointing to the H1 data directory.

```
{ 
    TChain chain("h42");
    chain.Add("$H1/dstarmb.root");
    //21330730 bytes, 21920 events
    chain.Add("$H1/dstarplb.root");
    //71464503 bytes, 73243 events
    chain.Add("$H1/dstarplb.root");
    //83827959 bytes, 85597 events
    chain.Add("$H1/dstarplb.root");
    //100675234 bytes, 103053 events
}
```

Run the above script from the command line:

```
root[] .x h1chain.C
```

**Step B:** Now we have a directory containing the four data files. Since a TChain is a descendent of TTree we can call TChain::Process to loop on all events in the chain. The parameter to the TChain::Process method is the name of the file containing the created TSelector class (h1analysis.C).

```
root[] chain.Process("h1analysis.C")
```

**Step C:** Same as step B, but in addition fill the event list with selected entries. The event list is saved to a file "elist.root" by the
TSelector::Terminate method. To see the list of selected events, you can do `elist->Print('all')`. The selection function has selected 7525 events out of the 283813 events in the chain of files. (2.65 per cent)

```
root[] chain.Process("h1analysis.C","fillList")
```

**Step D:** Process only entries in the event list. The event list is read from the file in `elist.root` generated by step C.

```
root[] chain.Process("h1analysis.C","useList")
```

**Step E:** The above steps have been executed with the interpreter. You can repeat the steps B, C, and D using ALCiC by replacing "h1analysis.C" by "h1analysis.C+" or "h1analysis.C++".

```
Step F: If you want to see the differences between the interpreter speed and ALCiC speed start a new session, create the chain as in step 1, then execute
```

The commands executed with the four different methods B, C, D and E produce two canvases shown below:

```
#include "h1analysis.h"
#include "TH2.h"
#include "TF1.h"
#include "TStyle.h"
#include "TCanvas.h"
#include "TLine.h"
#include "TEventList.h"
const Double_t dxbin = (0.17-0.13)/40; // Bin-width
const Double_t sigma = 0.0012;
TEventList *elist = 0;
Bool_t useList, fillList;
TH1F *hdmd;
TH2F *h2;
//_________________________________________________________
Double_t fdm5(Double_t *xx, Double_t *par)
{
    Double_t x = xx[0];
    if (x <= 0.13957) return 0;
    Double_t xp3 = (x-par[3])*(x-par[3]);
    Double_t res =
        dxbin*(par[0]*TMath::Power(x-0.13957, par[1])
    return res;
}
//_________________________________________________________
Double_t fdm2(Double_t *xx, Double_t *par)
{
    Double_t x = xx[0];
    if (x <= 0.13957) return 0;
    Double_t xp3 = (x-0.1454)*(x-0.1454);
    Double_t res =
        dxbin*(par[0]*TMath::Power(x-0.13957, par[1])
    return res;
}
//_________________________________________________________
void h1analysis::Begin(TTree *tree)
{
    function called before starting the event loop
    // -it performs some cleanup
    // -it creates histograms
    // -it sets some initialization for the event list
    //initialize the Tree branch addresses
    Init(tree);
    //print the option specified in the Process function.
    TString option = GetOption();
    printf("Starting h1analysis with process option:");
```
//Some cleanup in case this function had
//already been executed.
//Delete any previously generated histograms or
//functions
gDirectory->Delete("hdmd");
gDirectory->Delete("h2");
delete gROOT->GetFunction("f5");
delete gROOT->GetFunction("f2");

//create histograms
hdmd = new TH1F("hdmd","dm_d",40,0.13,0.17);
h2 = new TH2F
("h2","ptD0 vs dm_d",30,0.135,0.165,30,-3,6);

//process cases with event list
fillList = kFALSE;
useList  = kFALSE;
fChain->SetEventList(0);
delete gDirectory->GetList()->FindObject("elist");

// case when one creates/fills the event list
if (option.Contains("fillList")) {
  fillList = kTRUE;
  elist = new TEventList
  ("elist","selection from Cut",5000);
}

// case when one uses the event list generated
// in a previous call
if (option.Contains("useList")) {
  useList = kTRUE;
  TFile f("elist.root");
  elist = (TEventList*)f.Get("elist");
  if (elist) elist->SetDirectory(0);
}

// Otherwise the file destructor will delete elist
fChain->SetEventList(elist);
}

//_________________________________________________________
Bool_t h1analysis::ProcessCut(Int_t entry)
{
  // Selection function to select D* and D0.
  // In case one event list is given in input,
  // the selection has already been done.
  if (useList) return kTRUE;

  // Read only the necessary branches to select entries.
  // Return as soon as a bad entry is detected
  b_md0_d->GetEntry(entry);
  if (TMath::Abs(md0_d-1.8646) >= 0.04) return kFALSE;
  b_ptds_d->GetEntry(entry);
  if (ptds_d <= 2.5) return kFALSE;
  b_etads_d->GetEntry(entry);
  if (TMath::Abs(etads_d) >= 1.5) return kFALSE;
  b_ik->GetEntry(entry); ik--;
  // Original ik used f77 convention starting at 1
  b_ipi->GetEntry(entry); ipi--;
  //if nhitrp[ik]*nhitrp[ipi] <= 1 return kFALSE;
  //if rend[ik]-rstart[ik] <= 22 return kFALSE;
  //if ntracks <= 1 return kFALSE;
  //if njets < 1 return kFALSE;

  //if option fillList, fill the event list
  if (fillList) elist->Enter
  (fChain->GetChainEntryNumber(entry));
  return kTRUE;
}

//_________________________________________________________
void h1analysis::ProcessFill(Int_t entry)
{
  // Function called for selected entries only
  // Read branches not processed in ProcessCut
  b_dm_d->GetEntry(entry);
  b_rpd0_t->GetEntry(entry);
  b_ptd0_d->GetEntry(entry);

  // Fill some histograms
  hdmd->Fill(dm_d);
  h2->Fill(dm_d,rpd0_t/0.029979*1.8646/ptd0_d);
}

//_________________________________________________________
void h1analysis::Terminate()
{
  // Function called at the end of the event loop
  gStyle->SetOptFit();
  TCanvas *c1 = new TCanvas
  ("c1","h1analysis analysis",10,10,800,600);
  c1->SetBottomMargin(0.15);
  hdmd->SetAxisTitle("m_{K#pi#pi} - m_{K#pi}[GeV/c^{2}]");
  hdmd->GetAxis()->SetTitleOffset(1.4);
  hdmd->GetXaxis()->SetLabelSize(0.05);
  // Fit histogram hdmd with function f5 using
//the loglikelihood option
TF1 *f5 = new TF1("f5",fdm5,0.139,0.17,5);
f5->SetParameters(1000000, .25, 2000, .1454, .001);
hdmd->Fit("f5","l2");

//create the canvas for tau d0
gStyle->SetOptFit(0);
gStyle->SetOptStat(1100);
TCanvas *c2 = new TCanvas("c2","tauD0",100,100,800,600);
c2->SetGrid();
c2->SetBottomMargin(0.15);

// Project slices of 2-d histogram h2 along X,
// then fit each slice with function f2 and make a
// histogram for each fit parameter.
// Note that the generated histograms are added
// to the list of objects in the current directory.
TF1 *f2 = new TF1("f2",fdm2,0.139,0.17,2);
f2->SetParameters(1000, 10);
h2->FitSlicesX(f2,0,0,1,"qln");
TH1D *h2_1 = (TH1D*)gDirectory->Get("h2_1");
h2_1->SetXaxis()->SetTitle("#tau[ps]");
h2_1->SetMarkerStyle(21);
h2_1->Draw();
c2->Update();
TLine *line = new TLine(0,0,0,c2->GetUymax());
line->Draw();

// save the event list to a Root file if one was
// produced
if (fillList) {
    TFile efile("elist.root","recreate");
    elist->Write();
}
}
Networking

In this chapter, you will learn how to send data over the network using the ROOT socket classes.

Setting up a Connection

On the server side, we create a TServerSocket to wait for a connection request over the network. If the request is accepted, it returns a full-duplex socket. Once the connection is accepted, we can communicate to the client that we are ready to go by sending the string "go", and we can close the server socket.

```c++
// server
TServerSocket *ss = new TServerSocket(9090, kTRUE);
TSocket *socket = ss->Accept();
socket->Send("go");
ss->Close();
```

On the client side, we create a socket and ask the socket to receive input.

```c++
// client
TSocket *socket = new TSocket("localhost", 9090);
Char str[32];
Socket->Recv(str,32);
```

Sending Objects over the Network

We have just established a connection and you just saw how to send and receive a string with the example "go". Now let's send a histogram.

To send an object (in our case on the client side) it has to derive from TObject because it uses the Streamers to fill a buffer that is then sent over the connection. On the receiving side, the Streamers are used to read the object from the message sent via the socket. For network communication, we have a specialized TBuffer, a descendant of TBuffer called TMessage. In the following example, we create a TMessage with the intention to store an object, hence the constant kMESS_OBJECT in the constructor. We create and fill the histogram and write it into the message. Then we call TSocket::Send to send the message with the histogram.

```c++
// create an object to be sent
TH1F *hpx = new TH1F("hpx","px distribution",100,-4,4);
hpx->FillRandom("gaus",1000);
// create a TMessage to send the object
TMessage message(kMESS_OBJECT);
// write the histogram into the message buffer
message.WriteObject(hpx);
// send the message
socket->Send(message);
```

On the receiving end (in our case the server side), we write a while loop to wait and receive a message with a histogram. Once we have a message, we call TMessage::ReadObject, which returns a pointer to TObject. We have to cast it to a TH1 pointer, and now we have a histogram. At the end of the loop, the message is deleted, and another one is created at the beginning.

```c++
while (1) {
    TMessage *message;
    socket->Recv(message);
    TH1 *h = (TH1*)message->ReadObject(message->GetClass());
    delete message;
}
```
**Closing the Connection**

Once we are done sending objects, we close the connection by closing the sockets at both ends.

```c
Socket->Close();
```

This diagram summarizes the steps we just covered:

---

**A Server with Multiple Sockets**

Chances are that your server has to be able to receive data from multiple clients. The class we need for this is `TMonitor`. It lets you add sockets and the `TMonitor::Select` method returns the socket with data waiting. Sockets can be added, removed, or enabled and disabled.

Here is an example of a server that has a `TMonitor` to manage multiple sockets:

```c
TServerSocket *ss = new TServerSocket(9090, kTRUE);
// Accept a connection and return a full-duplex communication socket.
TSocket *s0 = ss->Accept();
TSocket *s1 = ss->Accept();

// tell the clients to start
s0->Send("go 0");
s1->Send("go 1");

// Close the server socket (unless we will use it // later to wait for another connection).
ss->Close();
TMonitor *mon = new TMonitor;
mon->Add(s0);
mon->Add(s1);

while (1) {
    TMessage *mess;
    TSocket  *s;
    s = mon->Select();
    s->Recv(mess);
    ...
}
```

The full code for the example above is in `$ROOTSYS/tutorials/hserver.cxx` and `$ROOTSYS/tutorials/hclient.cxx`. 
19 Writing a Graphical User Interface

The ROOT GUI classes support an extensive and rich set of widgets. The widgets classes depend only on the \texttt{X11} and \texttt{Xpm} libraries, eliminating the need for any other GUI engine such as Motif or QT, and they have the Windows look and feel. They are based on Hector Peraza's Xclass'95 widget library.

Although powerful and quite feature rich, we are missing extensive documentation. This will come eventually but for the time being you will have to "program by example". We start with a short tutorial followed by few non-trivial examples that will show how to use the different widget classes.

The New ROOT GUI Classes

Features of the new GUI classes in a nutshell:

- Originally based on Xclass'95 widget library (under a Lesser GNU Public License)
  - A rich and complete set of widgets
  - Uses only X11 and Xpm (no Motif, Xaw, Xt, etc.)
  - Small (12000 lines of C++)
  - Win'95 look and feel
- All X11 calls abstracted using the "abstract" ROOT TGXW class
- Rewritten to use internally the ROOT container classes
- Completely scriptable via the C++ interpreter (fast prototyping)
- Full class documentation is generated automatically (as for all ROOT classes)

XClass'95

Here are some highlights of the XClass'95. Hector Peraza is the original author of the XClass'95 class library.

The Xclass'95 comes with a complete set of widgets. These include:

- Simple widgets, as labels and icons
- Push buttons, either with text or pix maps
- Check buttons
- Radio buttons
- Menu bars and popup menus

The widgets are shown in frames:
frame, composite frame, main frame, transient frame, group frame

And arranged by layout managers:
horizontal layout, vertical layout, row layout, list layout, tile layout, matrix layout, ...

Using a combination of layout hints:
left, center x, right, top, center y, bottom, expand x, expand y and fixed offsets

Event handling by messaging (as opposed to callbacks): in response to actions widgets send messages (\texttt{SendMessage()}) to associated frames (\texttt{ProcessMessage()})

ROOT Integration

Replace all calls to X11 by calls to the ROOT abstract graphics base class \texttt{TGXW}. Currently, implementations of \texttt{TGXW} exist X11 (\texttt{TGX11}) and Win32 (\texttt{TGWIn32}). Thanks to this single graphics interface, porting ROOT to a new platform (BeOS, Rhapsody, etc.) requires only the implementation of \texttt{TGXW} (and \texttt{TSystem}).

Abstract Graphics Base Class \texttt{TGXW}
Concrete implementations of TGX are TGX11, for X Windows, TGWin32 for Win95/NT. The TGXClient implementation provides a network interface allowing for remote display via the rootdisp servers.

NOTE: the ROOT GUI classes are for the time being only supported on Unix/X11 systems. Work on a Win32 port is in progress and coming shortly

Further changes:

- Changed internals to use ROOT container classes, notably hash tables for fast lookup of frame and picture objects
- Added TObject inheritance to the few base classes to get access to the extended ROOT RTTI (type information and object inspection) and documentation system
- Conversion to the ROOT naming conventions to provide a homogeneous and consistent environment for the user

A Simple Example

The code that uses the GUI classes is written in bold font.

```c
#include <TApplication.h>
#include <TGClient.h>

int main(int argc, char **argv)
{
    TApplication theApp("App", &argc, argv);
    MyMainFrame mainWin(gClient->GetRoot(), 200, 220);
    theApp.Run();
    return 0;
}
```

MyMainFrame

```c
#include <TGClient.h>
#include <TOButton.h>
class MyMainFrame : public TGMainFrame {
private:
    TGTextButton *fButton1, *fButton2;
    TGPictureButton *fPicBut;
    TGCheckButton *fChkBut;
    TGRadioButton *fRBut1, *fRBut2;
    TGLayoutHints *fLayout;
public:
    MyMainFrame(const TGWindow *p, UInt_t w, UInt_t h);
    ~MyMainFrame();
    Bool_t ProcessMessage(Long_t msg, Long_t parm1, Long_t parm2);
};
```

Laying out the Frame

```c
MyMainFrame::MyMainFrame(const TGWindow *p, UInt_t w, UInt_t h): TGMainFrame(p, w, h)
{
    // Create a main frame with a number of different buttons.
    fButton1 = new TGTextButton(this, "&Version", 1);
    fButton1->SetCommand("printf("This is ROOT version %s\n",
gROOT->GetVersion());");
    fButton2 = new TGTextButton(this, "&Exit", 2);
    fButton2->SetCommand(".q" );
    fPicBut = new TGPictureButton( this, gClient->GetPicture("world.xpm"), 3);
    fPicBut->SetCommand("printf("hello world!\n");");
    fChkBut = new TGCheckButton(this, "Check Button", 4);
    fRBut1 = new TGRadioButton(this, "Radio Button 1", 5);
    fRBut2 = new TGRadioButton(this, "Radio Button 2", 6);
    fLayout = new TGLayoutHints
              (kLHintsCenterX | kLHintsCenterY);
    AddFrame(fButton1, fLayout);
    AddFrame(fPicBut, fLayout);
    AddFrame(fButton2, fLayout);
    AddFrame(fChkBut, fLayout);
    AddFrame(fRBut1, fLayout);
    AddFrame(fRBut2, fLayout);
    MapSubwindows();
    Layout();
    SetWindowName("Button Example");
    SetIconName("Button Example");
    MapWindow();
}```
Adding Actions

```cpp
Bool_t MyMainFrame::ProcessMessage(Long_t msg, Long_t parm1, Long_t parm2)
{
    // Process events generated by the buttons in the frame.
    switch (GET_MSG(msg)) {
    case kC_COMMAND:
        switch (GET_SUBMSG(msg)) {
        case kCM_BUTTON:
            printf("text button id %ld pressed\n", parm1);
            break;
        case kCM_CHECKBUTTON:
            printf("check button id %ld pressed\n", parm1);
            break;
        case kCM_RADIOBUTTON:
            if (parm1 == 5)
                fRBut2->SetState(kButtonUp);
            if (parm1 == 6)
                fRBut1->SetState(kButtonUp);
            printf("radio button id %ld pressed\n", parm1);
            break;
        default:
            break;
        }
    default:
        break;
    }
    return kTRUE;
}
```

The Result

![Graphic output of GUI](image)

The Widgets in Detail

In this section we look at an example of using the widgets. The complete source code is in `$ROOTSYS/test/guitest.C`. Build the test directory with the appropriate makefile, and you will be able to run guitest. Here we present snippets of the code and the graphical output.

First the main program, which reveals that the functionality is in `TestMainFrame`

```cpp
TROOT root("GUI", "GUI test environment");
int main(int argc, char **argv)
{
    TApplication theApp("App", &argc, argv);
    if (gROOT->IsBatch()) {
        fprintf(stderr,
            "%s: cannot run in batch mode\n", argv[0]);
        return 1;
    }
    TestMainFrame mainWindow(gClient->GetRoot(), 400, 220);
    theApp.Run();
    return 0;
}
```

`TestMainFrame` has two subframes (TGCompositeFrame), a canvas, a text entry field, a button, a menu bar, several popup menus, and layout hints. It has a public constructor, destructor and a `ProcessMessage` method to carry out the actions.

```cpp
class TestMainFrame : public TGMainFrame {
    private:
        TGCompositeFrame   *fStatusFrame;
        TGCanvas           *fCanvasWindow;
        TGCompositeFrame   *fContainer;
        TGTextEntry        *fTestText;
        TGButton           *fTestButton;
        TGMenuBar          *fMenuBar;
        TGPopupMenu        *fMenuFile, *fMenuTest, *fMenuHelp;
        TGPopupMenu        *fCascadeMenu,
            *fCascade1Menu, *fCascade2Menu;
        TGLayoutHints      *fMenuBarLayout, *fMenuBarItemLayout,
            *fMenuBarHelpLayout;
    public:
        TestMainFrame(const TGWindow *p, UInt_t w, UInt_t h);
        virtual ~TestMainFrame();
        virtual void CloseWindow();
        virtual Bool_t ProcessMessage(Long_t msg, Long_t parm1, Long_t parm2);
};
```

Example: Widgets and the Interpreter

The script `$ROOTSYS/tutorials/dialogs.C` shows how the widgets can be used from the interpreter.

```cpp
class TestMainFrame : public TGMainFrame {
    private:
        TGCompositeFrame *fStatusFrame;
        TGCanvas         *fCanvasWindow;
        TGCompositeFrame *fContainer;
        TGTextEntry      *fTestText;
        TGButton         *fTestButton;
        TGPopupMenu      *fMenuFile, *fMenuTest, *fMenuHelp;
        TGPopupMenu      *fCascadeMenu,
            *fCascade1Menu, *fCascade2Menu;
        TGLayoutHints    *fMenuBarLayout, *fMenuBarItemLayout,
            *fMenuBarHelpLayout;
    public:
        TestMainFrame(const TGWindow *p, UInt_t w, UInt_t h);
        virtual ~TestMainFrame();
        virtual void CloseWindow();
        virtual Bool_t ProcessMessage(Long_t msg, Long_t parm1, Long_t parm2);
};
```
RQuant Example

This is an example of extensive use of the ROOT GUI classes. I include only a picture here, for the curious the full documentation or RQuant can be found at: http://www.smartquant.com/welcome.html

References

http://home.cern.ch/~chytrace/xclassstut.html
A basic introduction and mini tutorial on the Xclass by Hector Peraza's
ac.be/html-test/xclass.html
The original Xclass'95 widget library documentation and source by Hector Peraza's.
http://www.smartquant.com/welcome.html
An Example of an elaborate ROOT GUI application.
20 Automatic HTML Documentation

The class descriptions on the ROOT website have been generated automatically by ROOT itself with the \texttt{THtml} class. With it, you can automatically generate (and update) a reference guide for your ROOT classes. Please read the \texttt{THtml} class description and the paragraph on Coding Conventions.

The following illustrates how to generate an html class description using the \texttt{MakeClass} method. In this example class name is \texttt{TBRIK}.

\begin{verbatim}
root[] THtml html; // instanciate a THtml object
root[] html->MakeClass("TBRIK")
\end{verbatim}

How to generate html code for all classes, including an index.

\begin{verbatim}
root[] html->MakeAll();
\end{verbatim}

This example shows how to convert a script to html, including the generation of a "gif" file produced by the script. First execute the script.

\begin{verbatim}
root[] .x htmlex.C
\end{verbatim}

Invoke the \texttt{TSystem} class to execute a shell script. Here we call the "\texttt{xpick}" program to capture the graphics window into a gif file.

\begin{verbatim}
root[] gSystem->Exec("xpick html/gif/shapes.gif")
\end{verbatim}

Convert this script into html.

\begin{verbatim}
root[] html->Convert("htmlex.C","Auto HTML document generation")
\end{verbatim}

For more details see the documentation of the class \texttt{THtml}. 
Building on the experience gained from the implementation and operation of the PIAF system we have developed the parallel ROOT facility, PROOF. The main problems with PIAF were because its proper parallel operation depended on a cluster of homogenous equally performing and equally loaded machines. Due to PIAF’s simplistic portioning of a job in N equal parts, where N is the number of processors, the overall performance was governed by the slowest node. The running of a PIAF cluster was an expensive operation since it required a cluster dedicated solely to PIAF. The cluster could not be used for other types of jobs without destroying the PIAF performance.

In the implementation of PROOF, we made the slave servers the active components that ask the master server for new work whenever they are ready. In the scheme the parallel processing performance is a function of the duration of each small job, packet, and the networking bandwidth and latency. Since the bandwidth and latency of a networked cluster are fixed the main tunable parameter in this scheme is the packet size. If the packet size is too small the parallelism will be destroyed by the communication overhead caused by the many packets sent over the network between the master and the slave servers. If the packet size is too large, the effect of the difference in performance of each node is not evened out sufficiently.

Another very important factor is the location of the data. In most cases, we want to analyze a large number of data files, which are distributed over the different nodes of the cluster. To group these files together we use a chain. A chain provides a single logical view of the many physical files. To optimize performance by preventing huge amounts of data being transferred over the network via NFS or any other means when analyzing a chain, we make sure that each slave server is assigned a packet, which is local to the node. Only when a slave has processed all its local data will it get packets assigned that cause remote access. A packet is a simple data structure of two numbers: begin event and number of events. The master server generates a packet when asked for by a slave server, taking into account the time it took to process the previous packet and which files in the chain are local to the slave server. The master keeps a list of all generated packets per slave, so in case a slave dies during processing, all its packets can be reprocessed by the left over slaves.
22 Threads

A thread is an independent flow of control that operates within the same address space as other independent flows of controls within a process. In most UNIX systems, thread and process characteristics are grouped into a single entity called a process. Sometimes, threads are called "lightweight processes".

Note: This introduction is adapted from the AIX 4.3 Programmer's Manual.

Threads and Processes

In traditional single-threaded process systems, a process has a set of properties. In multi-threaded systems, these properties are divided between processes and threads.

Process Properties

A process in a multi-threaded system is the changeable entity. It must be considered as an execution frame. It has all traditional process attributes, such as:

- Process ID, process group ID, user ID, and group ID
- Environment
- Working directory

A process also provides a common address space and common system resources:

- File descriptors
- Signal actions
- Shared libraries
- Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory)

Thread Properties

A thread is the schedulable entity. It has only those properties that are required to ensure its independent flow of control. These include the following properties:

- Stack
- Scheduling properties (such as policy or priority)
- Set of pending and blocked signals
- Some thread-specific data (TSD)

An example of thread-specific data is the error indicator, errno. In multi-threaded systems, errno is no longer a global variable, but usually a subroutine returning a thread-specific errno value. Some other systems may provide other implementations of errno.

With respect to ROOT, a thread specific data is for example the gPad pointer, which is treated in a different way, whether it is accessed from any thread or the main thread.

Threads within a process must not be considered as a group of processes (even though in Linux each thread receives an own process id, so that it can be scheduled by the kernel scheduler). All threads share the same address space. This means that two pointers having the same value in two threads refer to the same data. Also, if any thread changes one of the shared system resources, all threads within the process are affected. For example, if a thread closes a file, the file is closed for all threads.

The Initial Thread

When a process is created, one thread is automatically created. This thread is called the initial thread or the main thread. The initial thread executes the main routine in multi-threaded programs.

Thread Implementation in ROOT

The TThread class has been developed to provide a platform independent interface to threads for ROOT.

Installation

For the time being, it is still necessary to compile a threaded version of ROOT to enable some very special treatments of the canvas operations. We hope that this will become the default later.

To compile ROOT, just do (for example on a debian Linux):

```
./configure linuxdeb2 --with-thread=/usr/lib/libpthread.so
make depend
make
```
This configures and builds ROOT using /usr/lib/libpthread.so as the Pthread library, and defines R__THREAD. This enables the thread specific treatment of gPad, and creates $ROOTSYS/lib/libThread.so.

Note: The parameter linuxdeb2 has to be replaced with the appropriate ROOT keyword for your platform.

**Classes**

- **TThread**
  This class implements threads. The platform dependent implementation is in the TThreadImp class and its descendant classes (e.g. TPosixThread).

- **TMutex**
  This class implements mutex locks. A mutex is a mutually exclusive lock. The platform dependent implementation is in the TMutexImp class and its descendant classes (e.g. TPosixMutex).

- **TCondition**
  This class implements a condition variable. Use a condition variable to signal threads. The platform dependent implementation is in the TConditionImp class and its descendant classes (e.g. TPosixCondition).

- **TSemaphore**
  This class implements a counting semaphore. Use a semaphore to synchronize threads. The platform dependent implementation is in the TMutexImp and TConditionImp classes.

**TThread for Pedestrians**

To run a thread in ROOT, follow these steps:

**Initialization:**
Add these lines to your rootlogon.C:

```c
{
    // The next line may be unnecessary on some platforms
    gSystem->Load("/usr/lib/libpthread.so");
    gSystem->Load("$ROOTSYS/lib/libThread.so");
}
```

This loads the library with the TThread class and the pthread specific implementation file for Posix threads.

**Coding:**
Define a function (e.g. void* UserFun(void* UserArgs)) that should run as a thread. The code for the examples is at the web site of the authors (Jörn Adamczewski, Marc Hemberger). After downloading the code from this site, you can follow the example below.

www-linux.gsi.de/~go4/HOWTOthreads/howtothreadsbody.html#tth_sEc8

**Loading:**
Start an interactive ROOT session
Load the shared library:

```c
root [] gSystem->Load("mhs3.so");
```

Or

```c
root [] gSystem->Load("CalcPiThread.so");
```

**Creating:**
Create a thread instance (see also example RunMhs3.C or RunPi.C) with:

```c
root [] TThread *th = new TThread(UserFun, UserArgs);
```

When called from the interpreter, this gives the name “UserFun” to the thread. This name can be used to retrieve the thread later. However, when called from compiled code, this method does not give any name to the thread. So give a name to the thread in compiled use:

```c
root [] TThread *th = new TThread("MyThread", UserFun, UserArgs);
```

You can pass arguments to the thread function using the UserArgs-pointer.

When you want to start a method of a class as a thread, you have to give the pointer to the class instance as UserArg.

**Running:**

```c
root [] th->Run();
root [] TThread::Ps(); // like UNIX ps command;
```

With the mhs3 example, you should be able to see a canvas with two pads on it. Both pads keep histograms updated and filled by three different threads.

With the calcPi example, you should be able to see two threads calculating Pi with the given number of intervals as precision.
Asynchronous Actions

Different threads can work simultaneously with the same object. Some actions can be dangerous. For example, when two threads create a histogram object, ROOT allocates memory and puts them to the same collection. If it happens at the same time, the results are undetermined. To avoid this problem, the user has to synchronize these actions with:

\[ \text{TThread::Lock()} \]

// Locking the following part of code

\[ \text{TThread::Unlock()} \]

// Unlocking

The code between Lock() and UnLock() will be performed uninterrupted. No other threads can perform actions or access objects/collections while it is being executed. The TThread::Lock() and TThread::Unlock() methods internally use a global TMutex instance for locking. The user may also define his own TMutex instance and may locally protect his asynchronous actions by calling MyMutex.Lock() and MyMutex.Unlock().

Synchronous Actions: TCondition

To synchronize the actions of different threads you can use the TCondition class, which provides a signaling mechanism.

The TCondition instance must be accessible by all threads that need to use it, i.e. it should be a global object (or a member of the class which owns the threaded methods, see below). To create a TCondition object, a TMutex instance is required for the Wait and TimedWait locking methods. One can pass the address of an external mutex to the TCondition constructor:

\[ \text{TMutex MyMutex;}\]
\[ \text{TCondition MyCondition(MyMutex);}\]

If zero is passed, TCondition creates and uses its own internal mutex:

\[ \text{TCondition MyCondition();}\]

You can now use the following methods of synchronization:

- TCondition::Wait() waits until any thread sends a signal of the same condition instance: MyCondition.Wait() reacts on MyCondition.Signal() or MyCondition.Broadcast(). MyOtherCondition.Signal() has no effect.
- If several threads wait for the signal from the same TCondition MyCondition, at MyCondition.Signal() only one thread will react; to activate a further thread another MyCondition.Signal() is required, etc.
- If several threads wait for the signal from the same TCondition MyCondition, at MyCondition.Broadcast() all threads waiting for MyCondition are activated at once.

In some tests of MyCondition using an internal mutex, Broadcast() activated only one thread (probably depending whether MyCondition had been signaled before).

- MyCondition.TimedWait(secs,seconds) waits for MyCondition until the absolute time in seconds and nanoseconds since beginning of the epoch (January, 1st, 1970) is reached; to use relative timeouts “delta”, it is required to calculate the absolute time at the beginning of waiting “now”, for example:

\[
\text{Ulong_t now,then,delta;} \quad // \text{seconds}\\
\text{TDatime myTime;} \quad // \text{root daytime class}\\
\text{myTime.Set();} \quad // \text{myTime set to "now"}\\
\text{then=now+delta;} \quad // \text{absolute timeout}\\
\text{wait=MyCondition.TimedWait(then,0);} \quad // \text{waiting}
\]

- Return value wait of MyCondition.TimedWait should be 0, if MyCondition.Signal() was received, and should be nonzero, if timeout was reached.

The conditions example shows how three threaded functions are synchronized using TCondition: a ROOT script condstart.C starts the threads, which are defined in a shared library (conditions.cxx, conditions.h).

Xlib connections

Usually Xlib is not thread safe. This means that calls to the X could fail, when it receives X-messages from different threads. The actual result depends strongly on which version of Xlib has been installed on your system. The only thing we can do here within ROOT is calling a special function XInitThreads() (which is part of the Xlib), which should (!) prepare the Xlib for the usage with threads.

To avoid further problems within ROOT some redefinition of the gPad pointer was done (that’s the main reason for the recompilation). When a thread creates a TCanvas, this object is actually created in the main thread, this...
should be transparent to the user. Actions on the canvas are controlled via a function, which returns a pointer to either thread specific data (TSD) or the main thread pointer. This mechanism works currently only for gPad and will soon be implemented for other global Objects as e.g. gVirtualX, gDirectory, gFile.

Canceling a TThread

Canceling of a thread is a rather dangerous action. In TThread canceling is forbidden by default. The user can change this default by calling TThread::SetCancelOn(). There are two cancellation modes:

Deferred

Set by TThread::SetCancelDeferred() (default): When the user knows safe places in his code where a thread can be canceled without risk for the rest of the system, he can define these points by invoking TThread::CancelPoint(). Then, if a thread is canceled, the cancellation is deferred up to the call of TThread::CancelPoint() and then the thread is canceled safely. There are some default cancel points for pthreads implementation, e.g. any call of TCondition::Wait(), TCondition::TimedWait(), TThread::Join().

Asynchronous

Set by TThread::SetCancelAsynchronous(): If the user is sure that his application is cancel safe, he could call:

TThread::SetCancelAsynchronous();
TThread::SetCancelOn();
// Now cancelation in any point is allowed.
...
...
// Return to default
TThread::SetCancelOff();
TThread::SetCancelDeferred();

To cancel a thread TThread* th call:

th->Kill();

To cancel by thread name:

TThread::Kill(name);

To cancel a thread by ID:

TThread::Kill(tid);

To cancel a thread and delete th when cancel finished:

th->Delete();

Deleting of the thread instance by the operator delete is dangerous. Use th->Delete() instead. C++ delete is safe only if thread is not running.

Often during the canceling, some clean up actions must be taken. To define clean up functions use:

void UserCleanUp(void *arg){
   // here the user cleanup is done
   ...
}
TThread::CleanUpPush(&UserCleanUp,arg);
   // push user function into cleanup stack
   // "last in, first out"
TThread::CleanUpPop(1);
   // pop user function out of stack
   // and execute it,
   // thread resumes after this call
TThread::CleanUpPop(0);
   // pop user function out of stack
   // _without_ executing it

Note: CleanUpPush and CleanUpPop should be used as corresponding pairs like brackets; unlike pthreads cleanup stack (which is not implemented here), TThread does not force this usage.

Finishing thread

When a thread returns from a user function the thread is finished. It also can be finished by TThread::Exit(). Then, in case of pthread-detached mode, the thread vanishes completely.

By default, on finishing TThread executes the most recent cleanup function (CleanUpPop(1) is called automatically once).

Advanced TThread: Launching a Method in a Thread

Consider a class Myclass with a member function void*
Myclass::Thread0(void* arg) that shall be launched as a thread. To start Thread0 as a TThread, class Myclass may provide a method:

Int_t Myclass::Threadstart(){
   if(!mTh){
      mTh= new TThread("memberfunction",
            (void*) (void *)&Thread0,
            (void*) this);
      mTh->Run();
      return 0;
   }
   return 1;
}

Here mTh is a TThread* pointer which is member of Myclass and should be initialized to 0 in the constructor. The TThread constructor is called as when we used a plain C function above, except for the following two differences.

First, the member function Thread0 requires an explicit cast to (void*) (void *). This may cause an annoying but harmless compiler warning.
Strictly speaking, Thread0 must be a static member function to be called from a thread. Some compilers, for example gcc version 2.95.2, may not allow the (void*) (void*) is cast and just stop if Thread0 is not static. On the other hand, if Thread0 is static, no compiler warnings are generated at all.

Because the 'this' pointer is passed in 'arg' in the call to Thread0(void *arg), you have access to the instance of the class even if Thread0 is static. Using the 'this' pointer, non static members can still be read and written from Thread0, as long as you have provided Getter and Setter methods for these members.

For example:

```cpp
int_t state = arg->GetRunStatus();
arg->SetRunStatus(state);
```

Second, the pointer to the current instance of MyClass, i.e. (void*) *this, has to be passed as first argument of the threaded function Thread0 (C++ member functions internally expect the this pointer as first argument to have access to class members of the same instance). pthreads are made for simple C functions and do not know about Thread0 being a member function of a class. Thus, you have to pass this information by hand, if you want to access all members of the MyClass instance from the Thread0 function.

Note: Method Thread0 cannot be a virtual member function, since the cast of Thread0 to void (*) in the TThread constructor may raise problems with C++ virtual function table. However, Thread0 may call another virtual member function virtual void MyClass::Func0() which then can be overridden in a derived class of MyClass. (See example TMhs3).

Class MyClass may also provide a method to stop the running thread:

```cpp
int_t MyClass::Threadstop(){
    if(mTh){
        TThread::Delete(mTh);
        delete mTh;
        mTh=0;
        return 0;
    }
    return 1;
}
```

Example TMhs3: Class TThreadframe (TThreadframe.h, TThreadframe.cpp) is a simple example of a framework class managing up to four threaded methods. Class TMhs3(TMhs3.h, TMhs3.cpp) inherits from this base class, showing the mhs3 example 8.1 (mhs3.h, mhs3.cpp) within a class.

The Makefile of this example builds the shared libraries libTThreadframe.so and libTMhs3.so. These are either loaded or executed by the ROOT script TMhs3demo.C, or are linked against an executable TMhs3run.cxx.

### Known Problems

Parts of the ROOT framework, like the interpreter, are not yet thread-safe. Therefore, you should use this package with caution. If you restrict your threads to distinct and 'simple' duties, you will be able to benefit from their use.

The TThread class is available on all platforms, which provide a POSIX compliant thread implementation. On Linux, Xavier Leroy's Linux Threads implementation is widely used, but the TThread implementation should be usable on all platforms that provide pthread.

Linux Xlib on SMP machines is not yet thread-safe. This may cause crashes during threaded graphics operations; this problem is independent of ROOT.

**Object instantiation:** there is no implicit locking mechanism for memory allocation and global ROOT lists. The user has to explicitly protect his code when using them.

### Glossary

The following glossary is adapted from the description of the Rogue Wave Threads.h++ package.

#### Process

A process is a program that is loaded into memory and prepared for execution. Each process has a private address space. Processes begin with a single thread.

#### Thread

A thread of control, or more simply, a thread, is a sequence of instructions being executed in a program. A thread has a program counter and a private stack to keep track of local variables and return addresses. A multithreaded process is associated with one or more threads. Threads execute independently. All threads in a given process share the private address space of that process.

#### Concurrency

Concurrency exists when at least two threads are in progress at the same time. A system with only a single processor can support concurrency by switching execution contexts among multiple threads.

#### Parallelism

Parallelism arises when at least two threads are executing simultaneously. This requires a system with multiple processors. Parallelism implies concurrency, but not vice-versa.

#### Reentrant

A function is reentrant if it will behave correctly even if a thread of execution enters the function while one or more threads are already executing within the function. These could be the same thread, in the case of recursion, or different threads, in the case of concurrency.
Thread-specific data
Thread-specific data (TSD) is also known as thread-local storage (TLS). Normally, any data that has lifetime beyond the local variables on the thread's private stack are shared among all threads within the process. Thread-specific data is a form of static or global data that is maintained on a per-thread basis. That is, each thread gets its own private copy of the data.

Synchronization
Left to their own devices, threads execute independently. Synchronization is the work that must be done when there are, in fact, interdependencies that require some form of communication among threads. Synchronization tools include mutexes, semaphores, condition variables, and other variations on locking.

Critical Section
A critical section is a section of code that accesses a non-sharable resource. To ensure correct code, only one thread at a time may execute in a critical section. In other words, the section is not reentrant.

Mutex
A mutex, or mutual exclusion lock, is a synchronization object with two states locked and unlocked. A mutex is usually used to ensure that only one thread at a time executes some critical section of code. Before entering a critical section, a thread will attempt to lock the mutex, which guards that section. If the mutex is already locked, the thread will block until the mutex is unlocked, at which time it will lock the mutex, execute the critical section, and unlock the mutex upon leaving the critical section.

Semaphore
A semaphore is a synchronization mechanism that starts out initialized to some positive value. A thread may ask to wait on a semaphore in which case the thread blocks until the value of the semaphore is positive. At that time the semaphore count is decremented and the thread continues. When a thread releases semaphore, the semaphore count is incremented. Counting semaphores are useful for coordinating access to a limited pool of some resource.

Readers/Writer Lock
A multiple-readers, single-writer lock is one that allows simultaneous read access by many threads while restricting write access to only one thread at a time. When any thread holds the lock for reading, other threads can also acquire the lock reading. If one thread holds the lock for writing, or is waiting to acquire the lock for writing, other threads must wait to acquire the lock for either reading or writing.

Condition Variable
Use a condition variable in conjunction with a mutex lock to automatically block threads until a particular condition is true.

Multithread safe levels
A possible classification scheme to describe thread-safety of libraries:

- All public and protected functions are reentrant. The library provides protection against multiple threads trying to modify static and global data used within a library. The developer must explicitly lock access to objects shared between threads. No other thread can write to a locked object unless it is unlocked. The developer needs to lock local objects. The spirit, if not the letter of this definition, requires the user of the library only to be familiar with the semantic content of the objects in use. Locking access to objects that are being shared due to extra-semantic details of implementation (for example, copy-on-write) should remain the responsibility of the library.
- All public and protected functions are reentrant. The library provides protection against multiple threads trying to modify static and global data used within the library. The preferred way of providing this protection is to use mutex locks. The library also locks an object before writing to it. The developer is not required to explicitly lock or unlock a class object (static, global or local) to perform a single operation on the object. Note that even multithread safe level II hardly relieves the user of the library from the burden of locking.

Deadlock
A thread suffers from deadlock if it is blocked waiting for a condition that will never occur. Typically, this occurs when one thread needs to access a resource that is already locked by another thread, and that other thread is trying to access a resource that has already been locked by the first thread. In this situation, neither thread is able to progress; they are deadlocked.

Multiprocessor
A multiprocessor is a hardware system with multiple processors or multiple, simultaneous execution units.
List of Example files

Here is a list of the examples that you can find on the thread authors' web site (Jörn Adamczewski, Marc Hemberger) at:
www-linux.gsi.de/~go4/HOWTOthreads/howtothreadsbody.html#tth_sEc8

Example mhs3
- Makefile.mhs3
- mhs3.h
- mhs3LinkDef.h
- mhs3.cxx
- rootlogon.C
- RunMhs3.C

Example conditions
- Makefile.conditions
- conditions.h
- conditionsLinkDef.h
- conditions.cxx
- condstart.C

Example TMhs3
- Makefile.TMhs3
- TThreadframe.h
- TThreadframeLinkDef.h
- TThreadframe.cxx
- TMhs3.h
- TMhs3LinkDef.h
- TMhs3.cxx
- TMhs3run.cxx
- TMhs3demo.C

Example CalcPiThread
- Makefile.CalcPiThread
- CalcPiThread.h
- CalcPiThreadLinkDef.h
- CalcPiThread.cxx
- rootlogon.C
- RunPi.C
Appendix A: Install and Build ROOT

ROOT Copyright and Licensing Agreement:

This is a reprint of the copyright and licensing agreement of ROOT:

Copyright (C) 1995-2000, René Brun and Fons Rademakers.
All rights reserved.

ROOT Software Terms and Conditions

The authors hereby grant permission to use, copy, and distribute this software and its documentation for any purpose, provided that existing copyright notices are retained in all copies and that this notice is included verbatim in any distributions. Additionally, the authors grant permission to modify this software and its documentation for any purpose, provided that such modifications are not distributed without the explicit consent of the authors and that existing copyright notices are retained in all copies. Users of the software are asked to feed back problems, benefits, and/or suggestions about the software to the ROOT Development Team (rootdev@root.cern.ch). Support for this software - fixing of bugs, incorporation of new features - is done on a best effort basis. All bug fixes and enhancements will be made available under the same terms and conditions as the original software.

IN NO EVENT SHALL THE AUTHORS OR DISTRIBUTORS BE LIABLE TO ANY PARTY FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF THIS SOFTWARE, ITS DOCUMENTATION, OR ANY DERIVATIVES THEREOF, EVEN IF THE AUTHORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

THE AUTHORS AND DISTRIBUTORS SPECIFICALLY DISCLAIM ANY WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT. THIS SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, AND THE AUTHORS AND DISTRIBUTORS HAVE NO OBLIGATION TO PROVIDE MAINTENANCE, SUPPORT, UPDATES, ENHANCEMENTS, OR MODIFICATIONS.

Installing ROOT

To install ROOT you will need to go to the ROOT website at:
http://root.cern.ch/root/Availability.html

You have a choice to download the binaries or the source. The source is quicker to transfer since it is only 3.4 MB, but you will need to compile and link it. The binaries range from 7.4 MB to 11 MB depending on the target platform.

Choosing a Version

The ROOT developers follow the principle of "release early and release often", however a very large portion of a user base requires a stable product therefore generally three versions of the system is available for download - new, old and pro:

• The new version evolves quickly, with weekly or bi-weekly releases. Use this to get access to the latest and greatest, but it may not be stable. By trying out the new version you can help us converge quickly to a stable version that can then become the new pro version. If you are a new user we would advice you to try the new version.
• The pro (production) version is a version we feel comfortable with to exposing to a large audience for serious work. The change rate of this version is much lower than for the new version, it is about 3 to 6 months.
• The old version is the previous pro version that people might need for some time before switching the new pro version. The old change rate is the same as for pro.

Supported Platforms

For each of the three versions the full source is available for these platforms. Precompiled binaries are also provided for most of them:

• Intel x86 Linux (g++, egcs and KAI/KCC)
• Intel Itanium Linux (g++)
• HP Alpha-UX 10.x (HP CC and aCC, egcs1.1 C++ compilers)
• IBM AIX 4.1 (xlc compiler and egcs1.2)
• Sun Solaris for SPARC (SUN C++ compiler and egcs)
• Sun Solaris for x86 (SUN C++ compiler)
• Sun Solaris for x86 & KAI/KCC
• Compaq Alpha OSF1 (egcs1.2 and DEC/CXX)
• Compaq Alpha Linux (egcs1.2)
• SGI Irix (g++, KAI/KCC and SGI C++ compiler)
• Windows NT and Windows95 (Visual C++ compiler)
• Mac Mkmup and Linux PPC (g++)
• Hitachi HU-UX (egcs)
• LynxOS
• MacOS (CodeWarrior, no graphics)
Installing Precompiled Binaries

The binaries are available for downloading from

root.cern.ch/root/Availability.html.

Once downloaded you need to unzip and de-tar the file. For example, if you have downloaded ROOT v2.25 for HPUX:

% gunzip root_v2.25.00.HP-UX.B.10.20.tar.gz
% tar xvf root_v2.25.00.HP-UX.B.10.20.tar

This will create the directory root. Before getting started read the file README/README. Also, read the Introduction chapter for an explanation of the directory structure.

Installing the Source

You have a choice to download a compressed (tar ball) file containing the source, or you can login to the source code change control (CVS) system and check out the most recent source. The compressed file is a one time only choice; every time you would like to upgrade you will need to download the entire new version. Choosing the CVS option will allow you to get changes as they are submitted by the developers and you can stay up to date.

Installing and Building the source from a compressed file

To install the ROOT source you can download the tar file containing all the source files from the ROOT website. The first thing you should do is to get the latest version as a tar file. Unpack the source tar file, this creates directory 'root':

% tar xzvf root_v2.25.xx.source.tar.gz

Set $ROOTSYS to the directory where you want root to be installed:

% export ROOTSYS=<path>/root

Now type the build commands:

% cd root
% ./configure --help
% ./configure <target>
% gmake
% gmake install

Add $ROOTSYS/bin to PATH and $ROOTSYS/lib to LD_LIBRARY_PATH:

% export PATH=$ROOTSYS/bin:$PATH
% export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH

Try running root:

% root

It is also possible to setup and build ROOT in a fixed location. Please check README/INSTALL for more a detailed description of this procedure.

Target directory

By default, ROOT will be built in the $ROOTSYS directory. In that case the whole system (binaries, sources, tutorials, etc.) will be located under the $ROOTSYS directory.

Makefile targets

The Makefile is documented in details in the README/BUILDSYSTEM file. It explains the build options and targets.

More Build Options

To build the library providing thread support you need to define either the environment variable 'THREAD=-lpthread' or the configure flag '--with-threads=-lpthread' (it is the default for the linuxeseg architecture). [Note: this is only tested on Linux for the time being.]

To build the library providing CERN RFIO (remote I/O) support you need to define either the environment variable 'RFIO=<path>/libshift.a' or the configure flag '--with-rfio=<path>/libshift.a'. For pre-built version of libshift.a see ftp://root.cern.ch/root/shift/.

To build the PAW and Geant3 conversion programs b2root and g2root you need to define either the environment variable 'CERNLIB=<cernlibpath>' or the configure flag '--with-cern-libdir=<cernlibpath>'.

To build the MySQL interface library you need to install MySQL first. Visit http://www.mysql.com/ for the latest versions.

To build the strong authentication module used by rootd, you first have to install the SRP (Secure Remote Password) system. Visit http://jafar.stanford.edu/srp/index.html.

To build the event generator interfaces for Pythia and Pythia6, you first have to get the pythia libraries available from ftp: ftp://root.cern.ch/root/pythia/.

To use the libraries you have to define either the environment variable 'PYTHIA=<pythiadir>' or the configure flag '--with-pythia=<pythiadir>'.

Installing the Source from CVS

This paragraph describes how to checkout and build ROOT from CVS for Unix systems. For description of a checkout for other platforms, please see ROOT installation web page (http://root.cern.ch/root/CVS.html).

(Note: The syntax is for bash, if you use a t(csh) then you have to substitute export with setenv.)
% export CVSROOT=:pserver:cvs@root.cern.ch:/user/cvs
% cvs login
% (Logging in to cvs@root.cern.ch)
% CVS password: cvs
% cvs –z3 checkout root
  U root/…
  U –
% cd root
% ./configure --help
% ./configure <platform>
% gmake

If you are a part of a collaboration, you may need to use setup procedures specific to the particular development environment prior to running gmake.

You only need to run cvs login once. It will remember anonymous password in your $HOME/.cvspass file. For more install instructions and options, see the file README/INSTALL.

CVS for Windows
Although there exists a native version of CVS for Windows, we only support the build process under the Cygwin environment. You must have CVS version 1.10 or newer.

The checkout and build procedure is similar to that for Unix. For detailed install instructions, see the file README/INSTALL.

Converting a tar ball to a working CVS sandbox
You may want to consider downloading the source as a tar ball and converting it to CVS because it is faster to download the tar ball than checking out the entire source with CVS. Our source tar ball contains CVS information. If your tar ball is dated June 1, 2000 or later, it is already set up to talk to our public server (root.cern.ch). You just need to download and unpack the tar ball and then run following commands:

% cd root
% cvs -z3 update -d -P
% ./configure <platform>

Staying up-to-date
To keep your local ROOT source up-to-date with the CVS repository you should regularly run the command:
% cvs -z3 update -d -P

Setting the Environment Variables
Before you can run ROOT you need to set the environment variable ROOTSYS and change your path to include root/bin and library path variables to include root/lib. Please note: The syntax is for bash, if you are running tcsh you will have to use setenv and set instead of export.

1. Define the variable $ROOTSYS to the directory where you unpacked the ROOT:

% export ROOTSYS=/root

2. Add ROOTSYS/bin to your PATH:

% export PATH=$PATH:$ROOTSYS/bin

3. Set the Library Path
On HP-UX, before executing the interactive module, you must set the library path:
% export SHLIB_PATH=$SHLIB_PATH:$ROOTSYS/lib

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

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:

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

ROOTSYS is an environment variable pointing to the ROOT directory. For example, if you use the HPUX-10 AFS version you should set:
% export

To run the program just type: root

Documentation to Download

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

- ROOT Overview: Overview of the ROOT system (365 KB, 81 pages) ftp://root.cern.ch/root/ROOTMain.ps.gz
- ROOT Tutorials: The ROOT tutorials with graphics examples (320 KB, 81 pages) ftp://root.cern.ch/root/ROOTTutorials.ps.gz
- ROOT Classes: Description of all the ROOT classes (1.47 MB, 661 pages) ftp://root.cern.ch/root/ROOTClasses.ps.gz
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 (24 MB):
