Worked Examples in ROOT

Dan Clements
University of Glasgow

Contents

Chapter 1

Introduction

Welcome to 'Worked Examples in ROOT', I should first point out that this guide is unofficial and by no means a substitute for the official documentation that can be found at: 'http://root.cern.ch/root/doc/RootDoc.html'. The aim of this manual is to provide a shortened introduction to ROOT for new users who intend to analyse data, draw plots and work with ntuples. I will include lots of worked examples and try to keep the code and explanations as simple as possible.

The official manual describes ROOT thus:

> 'ROOT is an object-oriented framework aimed at solving the data analysis challenges of high-energy physics'

This is of course completely true, however I personally prefer to describe ROOT as:

> 'a collection of tools that allow me to draw graphs in C++'

In a very basic sense ROOT makes up for the fact that C++ does not have built in methods for drawing and saving graphs. ROOT also provides a good way of storing and accessing large amounts of data in files making it ideal for particle physics experiments. ROOT is implemented as a collection of 'objects'[1] that you can use within C++ to handle histogramming, storing data to file and analysis.

1.1 The many ways of running ROOT

Before we can get down to some worked examples and look at ROOT 'objects' we need to first explain the many and varied ways in which ROOT can be run. If you have ROOT installed on your computer and simply type 'root' at the command line then you will launch the 'Command Line Interface'[2]:

[1] Don't worry we'll describe 'objects' in the next chapter

[2] by adding the argument -l you can avoid the splash screen e.g: type: 'root -l'

```
[clements@ppepc10 clements]> root
  ****************************************
  *                                      *
  *        W E L C O M E  to  R O O T    *
  *                                      *
  *   Version   4.03/04      3 March 2006 *
  *                                      *
  *  You are welcome to visit our Web site *
  *          http://root.cern.ch         *
  *                                      *
  ****************************************

FreeType Engine v2.1.3 used to render TrueType fonts.
Compiled for linux with thread support.

CINT/ROOT C/C++ Interpreter version 5.15.169, Mar 14 2005
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.
root [0]
```

This interface allows you to open and inspect ROOT files and run ROOT scripts to perform analyses etc. The interface also allows ROOT objects to be created and manipulated, however this is better done by writing scripts in a text editor (like emacs, pico etc.) in most cases. The command line interface can be closed by issuing the command '.q'.

There are three main ways to run ROOT:

1. Via the CINT interactive interpreter.
2. Using a semi-compiled script with ACLiC.
3. From C++ linking the ROOT libraries externally.

We shall return to the relative merits of each method and examples of each in a later chapter. In this manual we are going to concentrate on using method 2 (ACLiC) and by means of introduction we are going to run the infamous 'Hello World' program by this method in our first worked example.

Open up your favourite text editor (e.g. emacs) and create a file 'wex1.C' with the following text:

```
\\Worked Example 1.1
#include<iostream>

using std::cout;
using std::endl;

int run(){
  cout<<"Hello World"<<endl;
  return 0;
}
```

In the same directory at the (Linux) command line then do:

```
[clements@ppepc10 ch1]> root -l
root [0] .L wex1.C++
Info in <TUnixSystem::ACLiC>: creating shared library /home/
clements/rootmanual/code/ch1/./wex1_C.so
root [1] run()
Hello World
(int)0
root [2]
```

If all goes well, then like above you should have 'Hello World' printed as above. The first line: 'root [0] .L wex1.C++' compiles the wex1.C script[3] you wrote whilst the second: 'root [1] run()' executes the specific function 'run()' you wrote within the script. The script 'wex1.C' is effectively C++ (or C to be more precise) except that a 'main()' function does not have to be defined. Instead the function you wish to execute is specified at the ROOT command line interface (i.e: 'root [1] run()').

Unless otherwise stated all the worked examples in this manual can be executed in a similar fashion as above. The only modification will be the script's name (e.g. 'wex1.C').

[3]the ++ denotes that you are using a C++ compiler

Chapter 2

Histograms

2.1 Introduction to ROOT Objects/Classes

ROOT is comprised of a number of 'objects', in C++ these are sometimes referred to as classes. A list of all these 'classes' can be found in the online documentation: http://root.cern.ch/root/Categories.html. An 'object' (or class) is a construct in the C++ programming language which can contain both data and functions. For example you might create a class to describe a cardboard box which would store its length, width and height along with a function that could calculate its volume. In the context of ROOT, the classes are things you'll want to use e.g. histograms, graphs, ntuples etc. To take advantage of ROOT all you need to do is learn what these classes can do and how to use them. It is not necessary to enter into the complexities of 'object oriented' design and the ROOT classes may be just as easily used in a C style, function based program.

2.2 Using a ROOT Histogram

ROOT contains a large number of classes, but to begin with let's consider a simple one-dimensional (TH1D) histogram. TH1D in ROOT is the name of the class which represents a 1-dimensional histogram[1], where each bin (or channel) is represented by a 'double precision' float.

Each class in ROOT has its own documentation which can be found by searching through the class categories at 'http://root.cern.ch/root/Categories.html' (Follow the blue links). In the case of TH1D histograms it can be found at: http://root.cern.ch/root/html/TH1D.html . It is worthwhile if you have access to the internet to spend a couple of minutes to take a look at this. The documentation begins with some instructions on how to use histograms, followed by a list of 'Function Members (methods)' and then 'Data Members'. The 'Function Members' perform actions on the histograms such as filling, integrating etc, whilst the data members record the state of the histogram (i.e. the bin values). As a rule when working with ROOT objects all the necessary control of the object is achieved by using the functions alone.

[1] This could be found through the online documentation http://root.cern.ch/root/Categories.html.

2.2.1 Basic Histogram Functions

If you have seen the histogram documentation you will notice that there are a lot of 'Function Members' to choose from. Finding the one you want can be a bit tricky at first so I am going to list the most basic ones:

```
Function Members (Methods)
public:
       TH1D(const char* name, const char* title, Int_t nbinsx,
            Double_t xlow, Double_t xup)
       virtual void TH1::Draw(Option_t* option = "")
       virtual Int_t TH1::Fill(Double_t x)
```

The functions are in order: the 'constructor' (TH1D), which creates an histogram object, a 'Draw' function which (you guessed) draws the histogram and a 'Fill' function for filling the histogram. Before we go any further its perhaps best to see these in action. Copy the code below into a file called wex2.1.C and run with ACLiC as shown in Chapter 1.

```
//Worked Example 2.1

#include<iostream>
#include "TH1D.h"

int run(){
  TH1D* histo=new TH1D("myhisto","myhisto",10,0,10); //constructor
  histo->Fill(5);
  histo->Draw();
  return 0;
}
```

If all went to plan, you should have arrived at a histogram which looks like Figure 2.1.

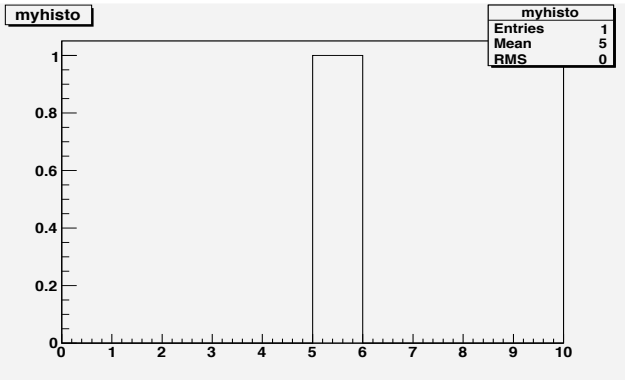

Figure 2.1: Histogram created by worked example 2.1.

The first feature of the code is the #include "TH1D.h" statement which is necessary as we are going to use the TH1D class[2]. The constructor is then called and a TH1D object is created with an associated pointer 'histo'. The arguments to the constructor follow that of the documentation and in this case create a histogram with name and title 'myhisto' with 10 bins, between 0 and 10 (constant bin width).

Once the constructor has been called all the member functions of the class can be called via the pointer (named 'histo' here) and the dereferencing operator ($\rightarrow$). In the example we use the 'Fill' and 'Draw' functions of the TH1D object 'histo'. Of course we could have chosen any of the functions given in the documentation and the ones you choose will depend on your analysis. The official documentation gives a good description (http://root.cern.ch/root/html/TH1D.html) of the available operations that includes adding histograms, normalising, copying etc. which I shall not attempt to emulate. However, I will include a few worked examples which shows how these may be used.

The first (Worked Example 2.2) shows how multiple histograms can be created. It is important to note that ROOT requires you to choose a different 'name' for every object you create with the constructor (i.e. histo1, histo2, histo3). The example then goes on to fill the histograms and perform basic histogram arithmetic. Finally, the resultant histogram is integrated and scaled.

[2]The relevant include statement for a given class is usually found in the top right corner in the official class documentation

```
//Worked Example 2.2
//Basic Histogram arithmetic and normalisation

#include<iostream>
#include "TH1D.h"

int run(){
  //Create 3 histograms using the constructors
  TH1D* histo1=new TH1D("histo1","histo1",10,0,10);
  TH1D* histo2=new TH1D("histo2","histo2",10,0,10);
  TH1D* histo3=new TH1D("histo3","histo3",10,0,10);

 //Fill the histograms
  histo1->Fill(1);
  histo2->Fill(3,2); //Fill with weight 2
  histo3->Fill(5);

  //Create an output histogram
  TH1D* histo4=new TH1D("histo4","histo4",10,0,10);

  //Add the input histograms to the output
  histo4->Add(histo1);
  histo4->Add(histo2);
  histo4->Add(histo3,-1); //subtract this one

  //Calculate the integral of the histogram
  cout<<"The integral of histo4 is:"<<histo4->Integral()<<endl;

  //Scale the histogram to give integral=1
  histo4->Scale(1.0/histo4->Integral());

  //Recalculate the integral
  cout<<"The new integral of histo4 is:"<<histo4->Integral()<<endl;

  //Draw the output histogram
  histo4->Draw();
  return 0;
}
```

The output histogram should look like Figure 2.2. The second example shows how a TH1D histogram can be created with user defined bin-widths by using an alternative constructor that takes an array (of doubles) to define the bin boundaries. Then an example is given of how the contents of a specific bin may be manipulated.

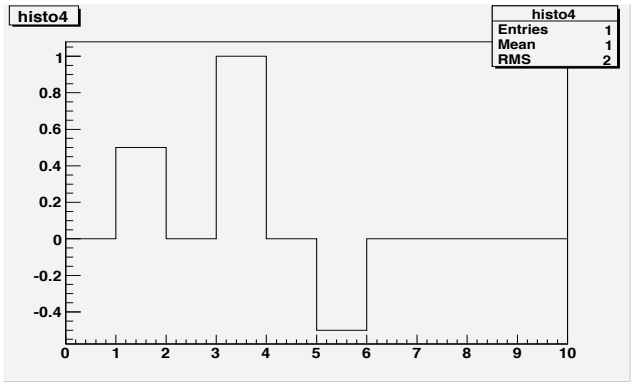

Figure 2.2: Histogram created by worked example 2.2.

```
//Worked Example 2.3
//Variable bin histograms and setting bin contents

#include<iostream>
#include "TH1D.h"

int run(){
  //Define a C array
  double array[]={1,3,5,9,13,17};

  //Define an histogram with variable bins using the array
  //The 5 relates to the number of bins.
  TH1D* histo=new TH1D("myhisto","myhisto",5,array);

  histo->Fill(4,2); //Fill with weight 2

  int binnum;
  //Find bin-number corresponding to x=15
  binnum=histo->FindBin(15);

  //Set the bin value of bin number 'binnum' to 3
  histo->SetBinContent(binnum,3); //Set content to 3

  //Draw the histogram
  histo->Draw();

  return 0;
}
```

The output histogram should look like Figure 2.3.

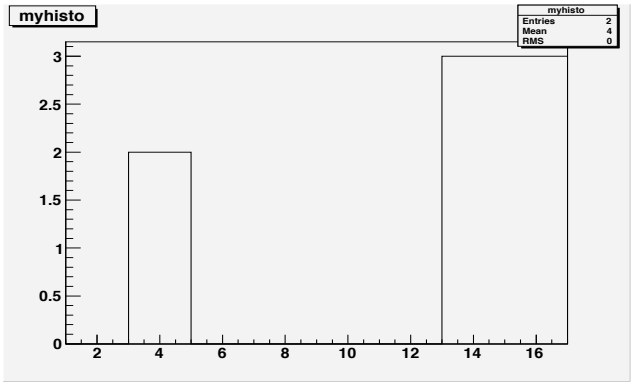

Figure 2.3: Histogram created by worked example 2.3.

2.2.2 Copying Histograms

You might what to create a transformed version of a histogram without destroying the original. This can be achieved by using the 'Clone' function. This is done in worked example 2.4. A histogram is created and then copied. The copy is then modified and drawn with the original on the same canvas.

```
//Worked Example 2.4
//Cloning a histogram and drawing both on the same axis

#include<iostream>
#include "TH1D.h"

int run(){
  //Define a histogram
  TH1D* histo=new TH1D("myhisto","myhisto",10,0,10);

  //Fill the histogram
  histo->Fill(5,1);
  histo->Fill(6,1);

  //Make a copy of this histogram
  TH1D* chisto=(TH1D*)histo->Clone("myclonehisto");

  //Modify the copy
  chisto->Scale(0.5);
  chisto->SetLineColor(2); //make the histo line colour red.

  //Draw both histograms on the same axis
  histo->Draw();
  chisto->Draw("same");

  return 0;
}
```

The output histogram should look like Figure 2.4:

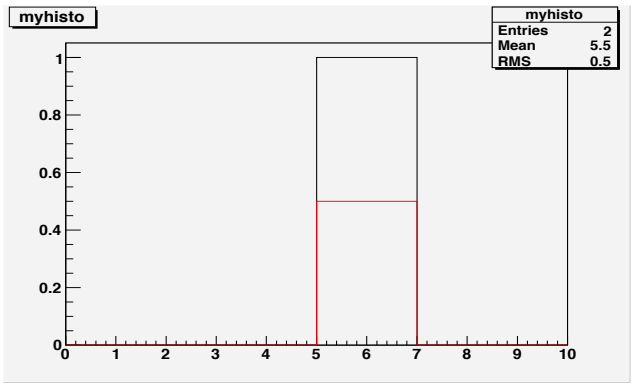

Figure 2.4: Histogram created by worked example 2.4

The actual line of code which performs the copying is a little complicated:

```
TH1D* chisto=(TH1D*)histo->Clone("myclonehisto");
```

If you look at the official documentation for the 'Clone' function of TH1D objects (http://root.cern.ch/root/html/TH1D.html) you will notice that it returns a TObject pointer (TObject*) and not a TH1D pointer. For this reason the pointer has to be explicitly 'cast' by applying (TH1D*). You will notice as well that the Clone function needs to be supplied with a new name (in this case 'myclonehisto'). This is because ROOT needs to assign a different name to each object in memory or it gets confused.

2.3 2D-Histograms and bin-finding

The TH1D class has a two dimensional equivalent called a TH2D class whise documentation can be found at: http://root.cern.ch/root/html/TH2D.html. In many ways this calss can be treated in an analogous way to TH2D but a few subtleties arise when trying to find and set specific bins as shown in the next example.

```
//Worked Example 2.5
//Two dimensional histograms and bin-finding

#include<iostream>
#include "TH2D.h"

using namespace std;

int run(){
  //Declare a 2D histogram
  TH2D* myhisto=new TH2D("myhisto","myhisto",10,0,10,20,0,20);

  //Fill the histogram
  myhisto->Fill(5,10,2);

  //Draw the histogram
  myhisto->Draw("LEGO2");

  //Find the bin content of the filled entry
  //First find the global bin-number
  int binnum=myhisto->FindBin(5,10);

  //Find this bin's content
  double binval=myhisto->GetBinContent(binnum);

  //Print these weights to the screen
  cout<<"The filled bin's global bin number is:"<<binnum<<endl;
  cout<<"The filled bin's weight is:"<<binval<<endl;

  return 0;
}
```

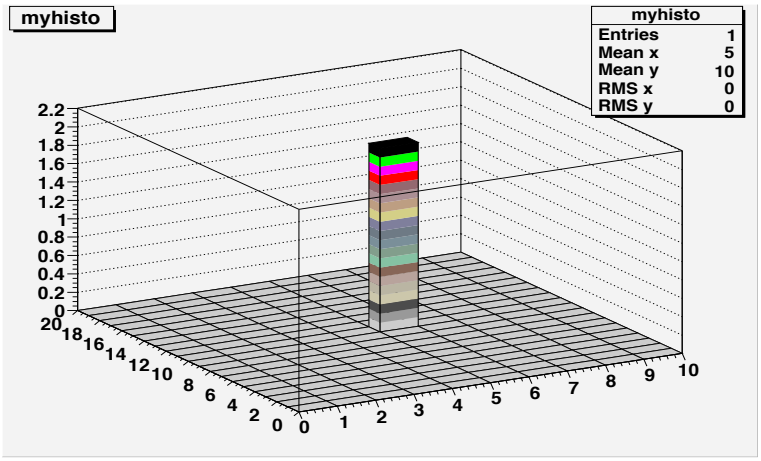

Figure 2.5: Histogram created by worked example 2.5

The example should produce the output as shown in Figure 2.5. In addition the ROOT command line should also give (amongst other things):

```
The filled bin's global bin number is:138
The filled bin's weight is:2
```

The TH2D object has its own include statement (#include "TH2D.h") which must be put in the preamble (as usual). The constructor for the object:

```
TH2D* myhisto=new TH2D("myhisto","myhisto",10,0,10,20,0,20);
```

follows that of the 1D case, except that the last 3 argument (20,0,20 here) relate to the y-axis parameters (no.of y-bins, y-minimum value, y-maximum value). The 'Fill' function also changes as the 'x' and 'y' co-ordinates must be given along with an optional 'weight':

```
myhisto->Fill(5,10,2);
```

In this case the bin at $x = 5$, $y = 10$ is being filled with a weight of 2; this can be verified by looking at the resultant histogram in Figure 2.5. To visualise a 2D histogram better it is sometimes best to change the drawing option, we used:

```
myhisto->Draw("LEGO2");
```

It is not necessary to add the argument "LEGO2" to the Draw function, but it does give us that nice 3D visualisation along with colours[3].

Two dimensional histograms sometimes create confusion with the use of the GetContent() and SetContent() functions. These methods can be used to manipulate the content of a bin and as we saw in previous example a command like:

```
myhisto->SetBinContent(5,3);
```

Will forcibly set the content of the 5th bin to a value of 3[4]. Similarly:

```
myhisto->GetBinContent(5);
```

will return the value (int this case a 'double precision float') of the fifth bin. In a two-dimensional histogram it can be less clear what the bin-number is. In order to provide a consistent interface ROOT gives every bin in a 2-D histogram a distinct number, just like the 1-D case. This global bin-number can be used to get and set the bin-content in exactly the same way as in 1-D. However, guessing the global bin-number (should you choose to do this) is trickier. For this reason its important to make use of the FindBin() command. In our example above we did:

```
int binnum=myhisto->FindBin(5,10);
```

To find the global bin-number of the bin corresponding to $x = 5$ and $y = 10$ (the bin we had previous filled). We can then use this number (stored in the variable binnum) in the GetBinContent command to return its weight:

[3]Other Draw()options can be found in the official ROOT manual.

[4]N.B. this is the fifth bin not x=5

```
double binval=myhisto->GetBinContent(binnum);
```

We could have equally done a SetBinContent(binnum,weight) to change the content of this bin to the value 'weight' ('weight' being a double precision float). The reason that the Global bin-number is not transparent in 2D histograms is that ROOT automatically defines an underlow and overflow bin for values outside the defined range in the constructor. In a 1-D histogram these can be accessed with global bin-numbers 0 and n+1 (where n is the number of bins defined in the constructor). In a 2-D histogram there has to be overflow available in two-dimensions and hence the numbering gets more complicated.

Chapter 3

Files

Clearly you will want to save and retrieve your ROOT histograms (or anything else) to files. The ROOT framework provides a nice interface for this, without having to deal with the sometimes complicated issues of 'streams' when using C++ alone. Like histograms, files are handled through a C++ class this time called a 'TFile' class[1]. As with every ROOT class, there is documentation: (http://root.cern.ch/root/html/TFile.html). Let's begin with creating a histogram and saving it to file:

```
// Worked Example 3.1
// Creating and saving a histogram to file

#include<iostream>
#include "TFile.h"
#include "TH1D.h"

int run(){
  //First create a histogram
  TH1D* histo=new TH1D("myhisto","myhisto",10,0,10);
  histo->Fill(5); //Fill once.

  //Open up a file
  TFile f1("outwex3.1.root","RECREATE"); //Open a file
  histo->Write();            //Write histogram to file

  return 0;

}
```

After running this script[2] you should find a file named 'outwex3.1.root' in the same directory as that which you ran the script from. The file object is created in the line:

```
TFile f1("outwex3.1.root","RECREATE"); //Open a file
```

[1] You've probably noticed a pattern in the naming of ROOT classes by now

[2] See Chapter 1 if you are unsure how to do this.

and is called 'f1'. The first argument 'outwex3.1.root' is the name of the file and the second argument is an option. The option 'RECREATE' means that the file will be newly created and will replace any existing file with the same name. The full list of options can be found with the class's official documentation. The other common choices are 'READ' (the default if no option is given) which opens a file for reading and 'UPDATE' which opens an existing file for writing whilst not overwriting its' contents. Once a file has been opened to save an object into it the 'Write' method has to be called (e.g. histo→Write()). The object is automatically stored to the currently open file.

There are two ways of instantiating objects in C++, as static objects or in the 'free store'[3]. The space for static objects is allocated at compilation time and results in an object whose member functions are reached through the '.' operator. This is the case in the 'TFile' object declared above. The object it creates is called 'f1' and the member functions could be reached via commands like 'f1.Close()'.

The alternative method (using the 'free store') is what we've been doing so far. It instantiates objects through commands like:

```
TH1D* histo=new TH1D("myhisto","myhisto",10,0,10);
```

Which creates a TH1D object with a pointer 'histo'. The member functions in this case are accessed through the (→) operator (e.g. histo→Fill(5)). The major difference in syntax of the constructor is the use of the 'new' operator. It is good programming practice to delete pointers after the object is no longer needed (see a C++ textbook for details).

It is possible to instantiate ROOT objects via either method, the 'free store' equivalent of the file opening line is:

```
TFile* f1=new TFile("outwex3.1.root","RECREATE");
```

This now creates a TFile pointer 'f1' whose member functions are now reached via the (→) operator (e.g. f1→Close()). Which method you choose is up to you. Generally pointers are easier to deal with in terms of passing to functions etc. and are ideal for objects such as histograms etc, however, care must be taken in some instances to reclaim the memory the objects use in the 'free store' (via the delete function) as problems can occur with 'memory leaks'. Static objects can be more cumbersome to manipulate and can run into difficulties if they go 'out of scope', but are generally free from 'memory leak' problems.

3.1 Retrieving objects

3.1.1 Via the command line interface

We can open the saved file through a script (like wex3.1.C) or directly from the interactive command-line interface. Usually you will want to write a script to do any useful work on saved data, however to simply look at saved histograms the interactive approach is sometimes better.

As usual the interface is opened by typing 'root' at the Linux command line. If this is done in the same directory as the saved ROOT file (outwex6.root) it can be opened by issuing the command:

[3]Sometimes referred to as the 'heap'

```
root [0] TFile f1("outwex3.1.root")
root [1] .ls
TFile**         outwex3.1.root
 TFile*         outwex3.1.root
  KEY: TH1D     myhisto;1       myhisto
root [2] myhisto.Draw()
```

These set of commands should allow you to access the saved histogram and draw it. The first command creates a TFile object (f1) from the already existing 'outwex6.root' file saved on disk[4] The '.ls' command lists the current file's contents which appear as a list of 'KEYs' which give the object type, name and title:

```
KEY: TH1D     myhisto;1       myhisto
```

You will notice that you were able to call the 'TH1D' member function 'Draw' on the 'myhisto' object without explicitly defining it. This is one of the features of CINT (the C interpreter). The name of objects in a loaded file can be used as a pseudo-object for use with the '.' or '→' operators. As CINT is an interpreted language the distinction between '.' and '→' becomes blurred.

There is an alternative method of looking at the contents of a ROOT file through a graphical 'TBrowser' interface:

```
root [0] TFile f1("outwex3.1.root")
root [1] TBrowser browser
```

This opens a new window (a TBrowser) which can be manipulated via the mouse (point and click). Click on 'ROOT files' and the filename should appear in the right hand pane. By clicking on the filename you can explore the contents of the file, and clicking on an object it can be drawn. It's worth taking a few minutes to explore some of the features of this interface. This method is particularly useful for exploring files of unknown structure (i.e. ones you didn't make yourself).

3.1.2 Via a script

Naturally you will also want to be able to open ROOT files and retrieve objects from scripts. An example of this is shown in worked example 3.2.

[4]Remember that if no second option is given to the constructor then the default is 'READ'.

```
//Worked Example 3.2
//Opening a file and retrieving an object from a script

#include<iostream>
#include "TFile.h"
#include "TH1D.h"

int run(){
  //Open the file
  TFile f1("outwex3.1.root");      //open the file for reading
  TH1D* histo;                    //define an histogram pointer
  f1.GetObject("myhisto",histo); //Get the object

  //Draw the object
  histo->SetDirectory(0);        //remove histogram from directory
  histo->Draw();

  return 0;
}
```

In this example, the histogram object created in worked example 6 is retrieved and drawn. The key line is:

```
f1.GetObject("myhisto",histo);
```

This gets the 'myhisto' object in the file 'f1' and points the previously defined TH1D pointer 'histo' to it. After this it is necessary to do a little fix:

```
histo->SetDirectory(0);
```

in order to prevent the object from disappearing when the file's directory is closed. This is a bit of a 'black magic' operation, but what it signifies is that you, as a user, are taking responsibility for the memory that the object takes (akin to the 'new' command). After this is done the 'histo' pointer can be treated as normal.

3.2 Structuring ROOT files

It is possible in your analysis that you will create a lot of histograms and will want to organise them within a ROOT file to sub-directories. An example of this is shown in worked example 3.2:

```
//Worked Example 3.3
//Organising a file into subdirectories

#include <iostream>
#include "TFile.h"
#include "TH1D.h"

int run(){
  //Create some histograms
  TH1D* histo1=new TH1D("histo1","histo1",10,0,10);
  TH1D* histo2=new TH1D("histo2","histo2",10,0,10);
  TH1D* histo3=new TH1D("histo3","histo3",10,0,10);

  //Fill the histograms
  histo1->Fill(3);
  histo2->Fill(5);
  histo3->Fill(7);

  //Open a file and make some subdirectories
  TFile f1("outwex3.3.root","RECREATE");
  f1.mkdir("mysubdir1");
  f1.mkdir("mysubdir2");

  f1.cd("/");
  histo1->Write();
  f1.cd("/mysubdir1");
  histo2->Write();
  f1.cd("/mysubdir2");
  histo3->Write();

  return 0;
}
```

The example when run should produce a file 'outwex8.root' with two subdirectories (mysubdir1 and mysubdir2). The file can be inspected using the TBrowser command shown earlier. If instead of the TBrowser you use the basic command line interface there are a few more commands to learn:

For an initiated TFile object 'f1' on the command line interface:

- f1.cd() return to root directory.
- f1.cd("/mysubdir1") switch to subdirectory 'mysubdir1'.
- .ls() list current directory.
- .pwd prints working directory.

Again you will want to access histograms in subdirectories via a script as shown in worked example 3.4:

```
//Worked Example 3.4
//Accessing objects from a file with subdirectories

#include<iostream>
#include "TFile.h"
#include "TH1D.h"

int run(){
  //Open the file
  TFile f1("outwex3.3.root");

  //Declare pointers
  TH1D *histop1, *histop2, *histop3;

  f1.GetObject("histo1",histop1);
  f1.GetObject("/mysubdir1/histo2",histop2);
  f1.GetObject("/mysubdir2/histo3",histop3);

  //Take responsibility for the histograms:
  histop1->SetDirectory(0);
  histop2->SetDirectory(0);
  histop3->SetDirectory(0);

  //Draw the histograms
  histop1->Draw();
  histop2->Draw("same");
  histop3->Draw("same");

  return 0;

}
```

3.3 Handling Multiple Files

It could be that the histograms you want to analyse are in more than one file. The same techniques can be used as is shown as above to obtain pointers to the objects. Firstly lets make two files each with a histogram called "myhisto", outwex3.5a.root and outwex3.5b.root.

```
//Worked Example 3.5
//Creating two files each with a histogram "myhisto"

#include<iostream>
#include "TFile.h"
#include "TH1D.h"

using namespace std;

int run(){
  //Declare histogram
  TH1D* histo=new TH1D("myhisto","myhisto",10,0,10);

  //Fill the histogram
  histo->Fill(5);

  //Open up two new files
  TFile* f1=new TFile("outwex3.5a.root","RECREATE");
  TFile* f2=new TFile("outwex3.5b.root","RECREATE");

  //Change to file f1 and write histogram
  f1->cd();
  histo->Write();

  //Change to file f2 and write histogram
  f2->cd();
  histo->Write();

  //Tidy up objects
  delete histo;
  delete f1;
  delete f2;

  return 0;
}
```

At the end of the above example you will notice that we delete the pointers which were instantiated using the 'new' command. This is good practice and we should have been doing it all along.

The difficulty occurs when we want to read both of the histograms again. The problem does not lie with the fact that there are two files, but rather that there are two histograms which share the same name 'myhisto'. As mentioned in Chapter 2, all ROOT objects in memory have to have a distinct name otherwise it gets confused. A way around this problem is to use the 'Clone' function as I will demonstrate below.

Often writing the entire program in a single function is not efficient so I will split the next worked example into two; the normal 'run()' and a separate function for handling the files. The program is still run in the normal manner (i.e. by calling the 'run()' function).

```
//Worked Example 3.6
//Reading histograms with the same name from different files

#include<iostream>
#include "TFile.h"
#include "TH1D.h"
#include "TString.h"

//Function that finds a histogram called "histoname" in the Tfile f1
//and copies to a cloned version named "newname". The cloned histogram
//is then returned. (N.B. only works for TH1D objects)

TH1D* GetHisto(TFile* f1, TString histoname, TString newname){

  //Declare a histogram (TH1D) pointer
  TH1D* histo;
  TH1D* outhisto;

  //Get the object from the file
  f1->GetObject(histoname,histo);

  //Check the object exists
  if(histo){
    outhisto=(TH1D*)histo->Clone(newname);
    outhisto->SetDirectory(0);
  }
  else{
    cout<<"No object called:"<<histoname<<" found in file."<<endl;
    outhisto=0; //Set to a null-pointer.
  }

  return outhisto;
}

int run(){
  //Open the files
  TFile* file1=new TFile("outwex3.5a.root");
  TFile* file2=new TFile("outwex3.5b.root");

  //Use our function 'GetHisto' to retrieve the histogram
  //from each file
  TH1D* histo1=GetHisto(file1,"myhisto","myhisto1");
  TH1D* histo2=GetHisto(file2,"myhisto","myhisto2");

  //Example continues on next page
```

```
  //Add the histograms
  histo1->Add(histo2);
  histo1->Draw();

  //Some tidying up.
  delete file1;
  delete file2;

  return 0;
}
```

If all goes well you should get the output as shown in Figure 3.1. This is as the script adds the two (identical) hisotgrams together.

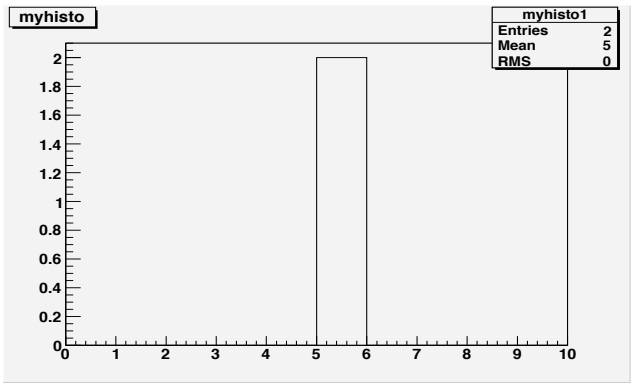

Figure 3.1: Histogram created by worked example 11.

A few things have been introduced here, the first is a new object called a 'TString'. You will notice the extra 'include' statement (#include "TString.h"). This class effectively works in the same way as the C++ string class. All ROOT names and titles can be handled with TStrings and they are very useful when writing functions as above.

The example is more complicated than those previous, but it's important to note that by splitting it into functions the code is easier to read and modify. At the end of the 'run()' function the TFile objects are deleted. We could have equally deleted the TH1D pointers 'histo1' and 'histo2' but this would have prevented us from viewing the results (of 'histo1-¿Draw') as the objects would have been lost from memory.

3.4 Converting a file from PAW

If you used PAW in the past to make your histograms you may have old HBOOK histogram files you want to convert to ROOT format. Fortunately there is a simple command line program to do this called 'h2root'. In the following example a HBOOK file 'pawexample.hbook' is converted to ROOT format. The output file created is simply 'pawexample.root'. The function should also be able to convert HBOOK ntuples into ROOT TTrees (see later). More information is given in the official documentation.

```
  [clements@ppepc10 ch3]> h2root pawexample.hbook
 RZOPEN. Cannot determine record length -  EXCHANGE mode is used.
 Converting directory //example
TFile**         pawexample.root HBOOK file:
pawexample.hbook converted to ROOT
 TFile*         pawexample.root HBOOK file:
pawexample.hbook converted to ROOT
  KEY: TH1F     h1001;1 dijet_1
  KEY: TH1F     h1002;1 dijet_2
  KEY: TH1F     h1003;1 dijet_3
  KEY: TH1F     h1004;1 dijet_4
  KEY: TH1F     h1010;1 phidif
```

3.5 Creating lots of histograms

It is possible you will want to create many histograms with the same bin-boundaries. As you will recall each individual histogram in ROOT must have a distinct name, however it is clearly not efficient to have to do this manually in some instances. To get around this a function can be written to convert an integer to a TString which may be used in a histogram title. This is shown in the example below which creates 100 histograms named histo0, histo1 ... histo99. The function that converts an integer to a TString is called ToString.

```
//Worked Example 3.7
//Creating a file with one hundred histograms

#include<iostream>
#include<sstream>
#include "TFile.h"
#include "TH1D.h"
#include "TString.h"

using namespace std;


//A function to convert an integer to a TString
TString ToString(int num) {
  ostringstream start;
  start<<num;
  TString start1=start.str();
  return start1;
}

//Example continues on next page
```

```
int run(){

  //Declare an array of histogram pointers
  TH1D* histos[100];

  //Start a loop and create the histograms
  for(int i=0; i<100; i++){

    //Convert the integer 'i' to a TString
    TString numstr=ToString(i);

    //Create the histogram's name
    TString histoname="histo"+numstr;

    //Declare the histogram
    histos[i]=new TH1D(histoname,histoname,100,0,100);

    //Fill the histogram
    histos[i]->Fill(i);
  }

  //Save all the histograms to a file
  TFile f1("outwex3.7.root","RECREATE");

  for(int i=0; i<100; i++){
    histos[i]->Write();
  }

  return 0;
}
```

The file containing the 100 histograms is called 'outwex3,7.root'. In this example you will observe that ROOT objects can be used in arrays (like the basic C++ types of int, double etc). The function 'ToString' converts its integer argument into a TString, an explanation of its form can be found in 'Ivor Horton's beginning C++' book, but the details aren't too important[5]. Note that you must include the line '#include¡sstream¿' in the preamble for the 'ToString' function to work.

3.6 Looping over a file's contents

On some occasions you may want to work with many objects stored in the file. Individually getting each one in the manner above can be come inefficient for large numbers and to avoid this ROOT offers a means of looping through a files contents non-interactively.

Taking the file created in 'Worked Example 3.7' with 100 histograms, the following

[5]I tend to just use the function as a 'black-box' recipe for performing the conversion.

example opens the file and loops over the histograms, and draws them together on the same axis. The result should be similar to that of Figure 3.2.

```
//Worked Example 3.8
//Looping over a files contents non-interactively

#include<iostream>
#include "TFile.h"
#include "TH1D.h"
#include "TKey.h"

using namespace std;

int run(){
  //Open a file for reading
  TFile* f1=new TFile("outwex3.7.root");
  int histocounter=0;

  //Create an iterator
  TIter nextkey( f1->GetListOfKeys() );
  TKey *key;

  //Loop over the keys in the file
  while ( (key = (TKey*)nextkey())) {

    // Read object from first source file
    TObject *obj = key->ReadObj();

    //Check that the object the key points to is a histogram.
    if ( obj->IsA()->InheritsFrom( "TH1" ) ) {
      histocounter++;

      //Cast the TObject pointer to a histogram one.
      TH1* histo = (TH1*)(obj);

      //Draw the histogram
      if(histocounter==1){
histo->Draw();
      }
      else{
histo->Draw("same");
      }
    }
  }
  return 0;
}
```

This example introduces the use of TKeys which can be used to iterate over a file's contents. The 'histocounter' lines are just a trick to allow us to draw all the histograms on the same set of axes. We will look further into iterators and containers in Chapter 6. This example is a lot more complicated than those previously but could be used as a recipe for iterating over histograms without going too much into the details.

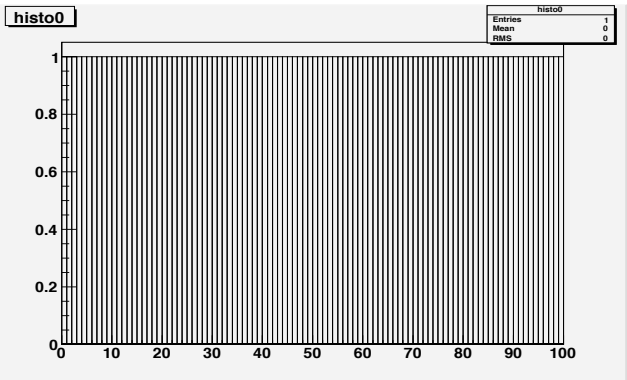

Figure 3.2: Histogram created by worked example 3.8

3.7 Merging histogram files using hadd

If you have a set of files that include histograms (with the same names) that you wish to merge this can be done using the 'hadd' routine. This can be called at the Linux command line and the relevant utility is generally stored as $ROOTSYS/bin/hadd. Simply typing 'hadd' at the command prompt gives instructions on how to use it:

```
[clements@ppepc10 clements]> hadd
Usage: hadd [-f] [-T] targetfile source1 source2 [source3 ...]
This program will add histograms from a list of root files and write them
to a target root file. The target file is newly created and must not
exist, or if -f ("force") is given, must not be one of the source files.
Supply at least two source files for this to make sense... ;-)
If the first argument is -T, Trees are not merged
[clements@ppepc10 clements]>
```

So for example to merge 'file1.root' and 'file2.root' and place the output in a file 'result.root':

```
[clements@ppepc10 clements]> hadd -T result.root file1.root file2.root
```

Chapter 4

Ntuples/Trees

4.1 Creating an ntuple

A lot of data analysis involves ntuples. Often these will come ready made from collaboration software in ROOT form. ROOT generally refers to ntuples as 'trees' and these are able to hold various forms of data as entries (e.g. arrays, vectors strings etc). The basic class that handles ROOT trees is called 'TTree' and in the following example we will create a simple ntuple and write the results to a file.

```
//Worked Example 4.1
//Creating a basic ntuple

#include<iostream>
#include "TTree.h"
#include "TMath.h"

int run(){

  //First open a file
  TFile f1("outwex4.1.root","RECREATE");

  //Create a TTree object
  TTree* tree=new TTree("tree","An example tree");

  float x,y,weight;

  //Create 3 branches
  tree->Branch("xval",&x,"x/F");
  tree->Branch("yval",&y,"y/F");
  tree->Branch("weight",&weight,"weight/F");

  //Example continues on next page
```

```
  //Generate values to fill tree
  for(int xval=0; xval<11; xval++){
    for(int yval=0; yval<11; yval++){
      //Determine x,y and weight
      x=(float)xval;
      y=(float)yval;
      weight=TMath::Gaus(x,5,1)*TMath::Gaus(y,5,1);

      //Fill the tree
      tree->Fill();
    }
  }

  //Save the file
  f1.Write();

  return 0;
}
```

This example creates a simple TTree with 3 branches which are all floats. These are xval, yval and weight. The TMath parts are just to provide the gaussian function (Gaus) we used to calculate the weight. There is a lot of flexibility in the way a Tree can be structured. The basic constructor just defines the name and title of the object:

```
TTree* tree=new TTree("tree","An example tree");
```

The details are defined within the branches which are declared in statements like:

```
tree->Branch("xval",&x,"x/F");
```

The first argument "xval" is the name of the branch. The second is an address to a variable or object (previously defined) which will be used to define the branch value[1]. The final argument defines the name and type of the variable in this case 'x', the '/F' denoting that the variable is a float. A full list of the possible variable types is given with the ROOT documentation.

You will notice that a file has to be opened before the TTree is declared. The reason for this is to prevent difficulties occurring in the event that the data contained within the TTree exceeds a computer's memory capacity. By having an open file the data can be stored away in a file before this becomes an issue.

4.2 Viewing Trees Interactively

ROOT provides a variety of methods for viewing the structire of TTrees interactively. The first is just using the TBrowser we encountered earlier for looking ar histograms. Using the result of the previous worked example 'outwex4.1.root':

[1]N.B. the & operator returns the address of a variable.

```
[clements@ppepc10 ch4]> root -l
root [0] TFile f1("outwex4.1.root")
root [1] TBrowser browser;
```

This launches the interactive browser, selecting 'ROOT Files' and then clicking on the name of the file 'outwex12.root' and then 'tree' gives a list of the branches. By clicking on a branch a histogram is drawn of its values over all entries.

The interactive methods are however more sophisticated than simple plotting of individual variables. A list of branches can be obtained directly by using the TTree's 'Print' command:

```
[clements@ppepc10 ch4]> root -l
root [0] TFile f1("outwex4.1.root")
root [1] .ls
TFile**         outwex4.1.root
 TFile*         outwex4.1.root
  KEY: TTree    tree;1  An example tree
root [2] tree->Print()
******************************************************************************
*Tree    :tree      : An example tree                                          *
*Entries :      121 : Total =            3622 bytes  File  Size =        821 *
*        :          : Tree compression factor =   1.00                       *
******************************************************************************
*Br    0 :xval      : x/F                                                      *
*Entries :      121 : Total  Size=       1095 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*.............................................................................*
*Br    1 :yval      : y/F                                                      *
*Entries :      121 : Total  Size=       1095 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*.............................................................................*
*Br    2 :weight    : weight/F                                                 *
*Entries :      121 : Total  Size=       1116 bytes  One basket in memory    *
*Baskets :        0 : Basket Size=      32000 bytes  Compression=   1.00     *
*.............................................................................*
root [3]
```

This also provides some information on the number of entries and their type. By using the TTree's 'Draw' command a fairly good insight can be made of the data. For instance:

```
root [3] tree->Draw("xval")
```

Draws the values of 'xval' in a one-dimensional histogram. Importantly though this 'Draw' command accepts basic cuts as a second argument e.g:

```
root [4] tree->Draw("xval","xval>5")
```

Will draw the same histogram but only considers points where 'xval' is greater than 5. This technique becomes powerful when considering more than one variable, e.g:

```
root [5] tree->Draw("weight:xval:yval")
```

Will draw a 3-dimensional histogram of 'weight' against 'xval' and 'yval'. This can be combined with a cut as in the one-dimensional case. There are a few more tricks that can be played and these are all well documented in the official guide to the TTree class (http://root.cern.ch/root/html/TTree.html) unser the 'Draw' member function. Its well worth taking a look at what's available as these methods tend to be quick to use and easy to adapt if there is a change in the underlying tree structure.

4.3 Reading Trees Non-Interactively - MakeClass

In many situations however it will be necessary to perform a tree (ntuple) analysis in a non-interactive way to make use of sophisticated cuts[2]. ROOT provides a framework for such analyses which reduces a lot of the basic defining of variables etc, which would otherwise be necessary to consider all the variables of the TTree being used. The method is 'MakeClass' and we will demonstrate how this works on our example above. The first thing to do is to call the 'MakeClass' method of the TTree you want to analyse:

```
 [clements@ppepc10 ch4]> root -l
root [0] TFile f1("outwex4.1.root")
root [1] .ls
TFile**         outwex4.1.root
 TFile*         outwex4.1.root
  KEY: TTree    tree;1  An example tree
root [2] tree->MakeClass("MyClass")
Info in <TTreePlayer::MakeClass>: Files: MyClass.h and MyClass.C
generated from TTree: tree
(Int_t)0
root [3]
```

The key line here is: tree-¿MakeClass("MyClass"). As a results of running this ROOT automatically creates two files one called 'MyClass.h' the other 'MyClass.C'. These contain the framework for an analysis over the TTree called 'tree' in this case. The files will appear in whatever directory you are currently running ROOT from.

The files can be a little confusing at first as they appear to contain quite a few methods, however to perform an analysis you will only have to modify a couple. The two files together define a 'C++' style class called 'MyClass', these behave in a similar way to the other classes we have encountered that make up ROOT. I'm going to list the files that been created and go through what each part does. You may find this easier to view using the actual source code in 'emacs' to take advantage of the syntax highlighting.

[2]These have a tendency to arise in many particle-physics applications.

MyClass.h File:

```
//////////////////////////////////////////////////////////
// This class has been automatically generated on
// Fri Jan 25 10:24:45 2008 by ROOT version 4.03/04
// from TTree tree/An example tree
// found on file: outwex4.1.root
//////////////////////////////////////////////////////////

#ifndef MyClass_h
#define MyClass_h

#include <TROOT.h>
#include <TChain.h>
#include <TFile.h>

class MyClass {
public :
   TTree          *fChain;   //!pointer to the analyzed TTree or TChain
   Int_t           fCurrent; //!current Tree number in a TChain

   // Declaration of leave types
   Float_t         xval;
   Float_t         yval;
   Float_t         weight;

   // List of branches
   TBranch        *b_x;   //!
   TBranch        *b_y;   //!
   TBranch        *b_weight;   //!

   MyClass(TTree *tree=0);
   virtual ~MyClass();
   virtual Int_t    Cut(Long64_t entry);
   virtual Int_t    GetEntry(Long64_t entry);
   virtual Long64_t LoadTree(Long64_t entry);
   virtual void     Init(TTree *tree);
   virtual void     Loop();
   virtual Bool_t   Notify();
   virtual void     Show(Long64_t entry = -1);
};

#endif
//Example continues on next page
```

```
#ifdef MyClass_cxx
MyClass::MyClass(TTree *tree)
{
// if parameter tree is not specified (or zero), connect the file
// used to generate this class and read the Tree.
   if (tree == 0) {
    TFile *f = (TFile*)gROOT->GetListOfFiles()->FindObject("outwex4.1.root");
      if (!f) {
         f = new TFile("outwex4.1.root");
      }
      tree = (TTree*)gDirectory->Get("tree");

   }
   Init(tree);
}

MyClass::~MyClass()
{
   if (!fChain) return;
   delete fChain->GetCurrentFile();
}

Int_t MyClass::GetEntry(Long64_t entry)
{
// Read contents of entry.
   if (!fChain) return 0;
   return fChain->GetEntry(entry);
}

Long64_t MyClass::LoadTree(Long64_t entry)
{
// Set the environment to read one entry
   if (!fChain) return -5;
   Long64_t centry = fChain->LoadTree(entry);
   if (centry < 0) return centry;
   if (fChain->IsA() != TChain::Class()) return centry;
   TChain *chain = (TChain*)fChain;
   if (chain->GetTreeNumber() != fCurrent) {
      fCurrent = chain->GetTreeNumber();
      Notify();
   }
   return centry;
}
//Example continues on next page
```

```
void MyClass::Init(TTree *tree)
{
   // The Init() function is called when the selector needs to initialize
   // a new tree or chain. Typically here the branch addresses of the tree
   // will be set. It is normaly not necessary to make changes to the
   // generated code, but the routine can be extended by the user if needed.
   // Init() will be called many times when running with PROOF.

   // Set branch addresses
   if (tree == 0) return;
   fChain = tree;
   fCurrent = -1;
   fChain->SetMakeClass(1);

   fChain->SetBranchAddress("xval",&xval);
   fChain->SetBranchAddress("yval",&yval);
   fChain->SetBranchAddress("weight",&weight);
   Notify();
}

Bool_t MyClass::Notify()
{
   // The Notify() function is called when a new file is opened. This
   // can be either for a new TTree in a TChain or when when a new TTree
   // is started when using PROOF. Typically here the branch pointers
   // will be retrieved. It is normaly not necessary to make changes
   // to the generated code, but the routine can be extended by the
   // user if needed.

   // Get branch pointers
   b_x = fChain->GetBranch("xval");
   b_y = fChain->GetBranch("yval");
   b_weight = fChain->GetBranch("weight");

   return kTRUE;
}

void MyClass::Show(Long64_t entry)
{
// Print contents of entry.
// If entry is not specified, print current entry
   if (!fChain) return;
   fChain->Show(entry);
}

//Example continues on next page
```

```
Int_t MyClass::Cut(Long64_t entry)
{
// This function may be called from Loop.
// returns  1 if entry is accepted.
// returns -1 otherwise.
   return 1;
}
#endif // #ifdef MyClass_cxx
```

Now the good news is that if the structure of the TTree remains unchanged (i.e. the same variable names and types) there is often little need to change this file. Those familiar with C++, will recognise the basic building blocks of a class including public variables, methods a constructor and a destructor. The functions defined above serve the purpose of loading a TTree from a file and initialising all the variables. The actual analysis part takes place in the MyClass.C file:

```
MyClass.C file:

#define MyClass_cxx
#include "MyClass.h"
#include <TH2.h>
#include <TStyle.h>
#include <TCanvas.h>

void MyClass::Loop()
{
//   In a ROOT session, you can do:
//      Root > .L MyClass.C
//      Root > MyClass t
//      Root > t.GetEntry(12); // Fill t data members with entry number 12
//      Root > t.Show();       // Show values of entry 12
//      Root > t.Show(16);     // Read and show values of entry 16
//      Root > t.Loop();       // Loop on all entries
//

//     This is the loop skeleton where:
//    jentry is the global entry number in the chain
//    ientry is the entry number in the current Tree
//  Note that the argument to GetEntry must be:
//    jentry for TChain::GetEntry
//    ientry for TTree::GetEntry and TBranch::GetEntry
//
//       To read only selected branches, Insert statements like:
// METHOD1:
//    fChain->SetBranchStatus("*",0);  // disable all branches
//    fChain->SetBranchStatus("branchname",1);  // activate branchname
// METHOD2: replace line
//    fChain->GetEntry(jentry);       //read all branches
//by  b_branchname->GetEntry(ientry); //read only this branch
   if (fChain == 0) return;

   Long64_t nentries = fChain->GetEntriesFast();

   Int_t nbytes = 0, nb = 0;
   for (Long64_t jentry=0; jentry<nentries;jentry++) {
      Long64_t ientry = LoadTree(jentry);
      if (ientry < 0) break;
      nb = fChain->GetEntry(jentry);   nbytes += nb;
      // if (Cut(ientry) < 0) continue;
   }
}
```

As you will see the automatically generated file contains quite a few comments to

describe how to use it. However these rely on the CINT interpreter which for reasons I will describe later is not always the best approach. In order to get the framework to work with compiled libraries we need to change the Loop() function above an add another called 'run()' to control things. The changes to MyClass.C are shown below:

```
//Modified MyClass.C File (for use with ACLiC)
#define MyClass_cxx
#include "MyClass.h"
#include <TH2.h>
#include <TStyle.h>
#include <TCanvas.h>
#include<iostream>

using namespace std;

void MyClass::Loop()
{
  Long64_t nentries = fChain->GetEntries();

  for(int jentry=0; jentry<nentries;jentry++) {
    GetEntry(jentry);
    cout<<xval<<" "<<yval<<" "<<weight<<endl;
  }
}

//The overall controlling function
int run(){
  //Instantiate a MyClass object
  MyClass m;

  //Run the object's Loop() function
  m.Loop();

  return 0;
}
```

After these changes can be made the analysis can be run by simply doing:

```
[clements@ppepc10 ch4]> root -l
root [0] .L MyClass.C++
Info in <TUnixSystem::ACLiC>: creating shared library /home/clements/
rootmanual/code/ch4/./MyClass_C.so
In file included from /home/clements/rootmanual/code/ch4/MyClass.C:2,
                 from /home/clements/rootmanual/code/ch4/filegUnSAs.h:32,
                 from /home/clements/rootmanual/code/ch4/filegUnSAs.cxx:16:
/home/clements/rootmanual/code/ch4/MyClass.h: In member function
‘virtual Int_t
   MyClass::Cut(long long int)’:
/home/clements/rootmanual/code/ch4/MyClass.h:131: warning: unused parameter ‘
   Long64_t entry’
root [1] run()
```

If all goes well this should print out the x, y and weight values for each entry. The code that actually prints the values is in the Loop() function which actually performs the analysis:

```
void MyClass::Loop()
{
  Long64_t nentries = fChain->GetEntries();

  for(int jentry=0; jentry<nentries;jentry++) {
    GetEntry(jentry);
    cout<<xval<<" "<<yval<<" "<<weight<<endl;
  }
}
```

This function first finds out how many entries there are in the Tree using the GetEntries function and then proceeds to loop over each one in the ‘for’ loop. The function GetEntry() is defined in the header file MyClass.h and collects the value of each variable in the Tree for the entry given by the argument (jentry inthis case). This means that the values ‘xval’, ‘yval’ and ‘weight’ (also defined in the MyClass.h file) take on their correct values for the given entry. The TTree’s members (e.g. xval, yval and weight) can then be used in any way the user desires (in this case simply printing them to the screen).

It is possible to add extra function like Loop() in order to develop a more sophisticated analysis program. In order for them to be a member of the class and have access to all the public variables associated in the TTree however, they must be defined in MyClass.h in a similar fashion.

4.4 Running the same analysis on a different file

It might be the case that you want to run your TTree analysis over a tree that is different from the one you originally used to create your framework files (e.g. MyClass.C, MyClass.h). If you look back to MyClass.h and the MyClass constructor ‘MyClass::MyClass(TTree *tree)’ you will notice that the file that the data comes from ‘out-

wex14.root' is hardwired as a default option. This means that the TTree analysis will look for data in this this file if no other is specified.

We can specify the file to be used in the controlling run() function we added to MyClass.C. I.e to read in a TTree with the same structure as before called 'tree' from a file 'outwex14b.root' the following changed to the run() function in MyClass.C are made:

```
  int run(){
  //Instantiate a MyClass object
  TFile f1("outwex4.1b.root");
  TTree* tree;
  f1.GetObject("tree",tree);
  MyClass m(tree);
  m.Init(tree);

  //Run the object's Loop() function
  m.Loop();

  delete tree;
  cout<<"Program finished"<<endl;

  return 0;
}
```

This opens up a file, gets a pointer to the TTree object and feeds it to the MyClass constructor. Care must be taken to delete the TTree object after use as this can lead to program failures.

4.5 Running an existing analysis on a Chain

TTree objects can be chained together in the event of more data being made available. This can also be taken into account for an existing analysis framework by modifying the run() function in MyClass.C:

```
int run(){
  //Create a TChain Object
  TChain* chain=new TChain("tree");
  chain->Add("outwex4.1.root");
  chain->Add("outwex4.1b.root");
  MyClass m(chain);
  m.Init(chain);

  //Run the object's Loop() function
  m.Loop();

  delete chain;
  cout<<"Program finished"<<endl;

  return 0;
}
```

This first creates a TChain of the "tree" TTree from two files 'outwex4.1.root' and 'outwex4.1b.root'. The TChain is then passed to the MyClass constructor instead of the TTree.

4.6 A More Complicated Tree - Vectors

Unfortunately not all TTrees are as simple as that used in the previous example. One particular problem that is sometimes encountered occurs when a more complicated object such as a C++ vector is included as a TTree branch. The MakeClass function on such a TTree will still make the ".C" and ".h" files as before but may need a little modification before they can work.

If the TTree members in the header file appear as below, remember that for instance PtGen is a pointer to vector and needs to be dereferenced to return to the actual object.

```
vector<float>   *PtGen;
vector<float>   *PhiGen;
vector<float>   *EtaGen;
```

You may find that you have to add "#include<vector>" statements and a "using namespace std" to the analysis header '.h' and '.C' files.

Chapter 5

Making Histograms Look Better

Its possible you will want to make aesthetic changes to your histograms in order to impress your fellow physicists. Some of the things that you can do have been shown in previous examples, but we'll go over them again for the sake of destroying trees[1].

5.1 Plotting Multiple Histograms On One Set Of Axis

In worked example 5.1, we simply create two histograms, and draw them on the same axis:

```
//Worked Example 5.1
//Plotting Multiple Histograms On A Single Set Of Axis

#include<iostream>
#include "TH1D.h"

using namespace std;

int run(){
  //Create two histograms
  TH1D* histo1=new TH1D("myhisto1","myhisto1",10,0,10);
  TH1D* histo2=new TH1D("myhisto2","myhisto2",10,0,10);

  //Fill the histogram
  histo1->Fill(5);
  histo2->Fill(7);

  //Change the line colour of the second histogram
  histo2->SetLineColor(2);

  //Example continues on next page
```

[1] And hopefully contributing to Global Warming in the process.

```
  //Draw First plot
  histo1->Draw();

  //Draw Second plot using the same axis
  histo2->Draw("same");

  return 0;
}
```

The option "same" given to the Draw command instructs ROOT to plot the histogram on the same axis (TCanvas) as the previously drawn object. It is not necessary to change the colour of the second histogram, but it can help to tell the two histograms apart.

A problem can arise however if the second histogram has a bin, which contains a weight much higher than the previously drawn one. ROOT has to decide the minimum and maximum y-values to display and it does this (when using the Draw command) by taking into account the minimum and maximum weights across all bins. However, if we use the "same" command we effectively instruct ROOT to plot the histogram on the existing TCanvas without such considerations.

As a result it is possible that the second histogram (the one which is drawn using "same") will have bins which exceed the displayed y-axis limits and effectively be cut off. This tends to make histograms look quite poor, but as always ROOT has an answer.

5.2 Adjusting Displayed Range Of Histograms

The histogram ROOT class has a function that can change the displayed ranges of the x and y axis. Techinically speaking though the function actually belongs to the TAxis class which is part of the TH1D class. For a TH1D pointer called 'histo', the x-axis may be changed by:

```
histo->GetXaxis()->SetRangeUser(0,5);
```

In this case the x-axis of the histogram (whose pointer is histo) will be set from 0 to 5. In a similar fashion the y-axis may be adjusted:

```
histo->GetYaxis()->SetRangeUser(2,4);
```

In this the case the y-axis limits being set between 2 and 4. The construction of the command to change the axis limits might seem a little bit complicated at first and it results from the C++ nature of ROOT. ROOT is a set of classes, that is a set of self-contained objects with data and member functions. In this design large complicated objects like histograms can be made from a number of smaller less complicated ones. For the histogram class the axis and their controlling functions are distinct classes. In order to change the axis property we must first get a pointer to the axis subobject of the histogram (histo-¿GetYaxis()) and then apply the desired function to this. The SetRangeUSer function is part of the TAxis class (http://root.cern.ch/root/html/TAxis.html).

5.3 Plotting Multiple Histograns On Different Axis

Occassionally you might want to draw lots of hisotgrams separately on different axis but close together (e.g. to create a single .eps file). This can be achieved by using a multi-panel canvas:

```
  //Worked Example 5.2
//Using a multiple-panel canvas to plot hiostograms

#include<iostream>
#include "TH1D.h"
#include "TCanvas.h"

int run(){
  //Create some histograms
  TH1D* histo1=new TH1D("histo1","histo1",10,0,10);
  TH1D* histo2=new TH1D("histo2","histo2",10,0,10);
  TH1D* histo3=new TH1D("histo3","histo3",10,0,10);
  TH1D* histo4=new TH1D("histo4","histo4",10,0,10);

  //Fill the histograms
  histo1->Fill(2);
  histo2->Fill(3);
  histo3->Fill(5);
  histo4->Fill(8);

  //Create a canvas
  TCanvas *mycanvas=new TCanvas("mycanvas","A Set Of Histos",1);

  //Split the canvas into a 2 * 2 grid
  mycanvas->Divide(2,2);

  //Select the first pane
  mycanvas->cd(1);
  histo1->Draw();

  //Select the second pane
  mycanvas->cd(2);
  histo2->Draw();

  //Select the third pane
  mycanvas->cd(3);
  histo3->Draw();

  //Example continues on next page
```

```
  //Select the fourth pane
  mycanvas->cd(4);
  histo4->Draw();

  return 0;
}
```

If all goes well the output should look like Figure 5.1:

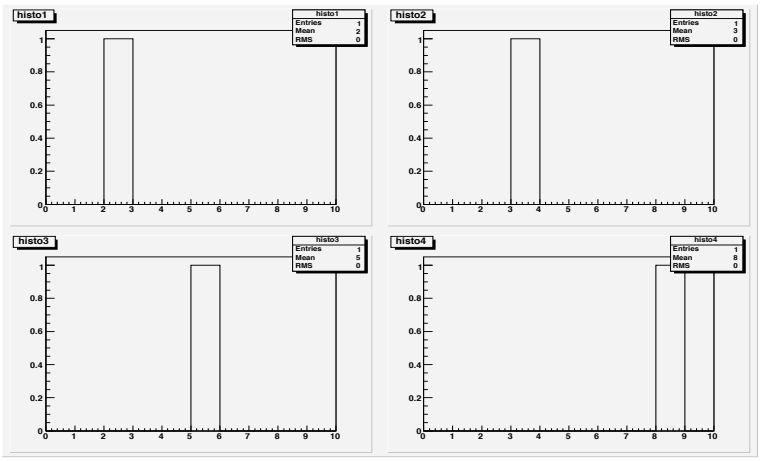

Figure 5.1: Histogram created by worked example 5.2.

The new ingredient in this example is the TCanvas class, which like the other must be made available by use of an 'include' statement (#include "TCanvas.h"). The TCanvas class documentation is found at: http://root.cern.ch/root/html/TCanvas.html and there are a lot of options for customising their appearance.

The TCanvas contructor first declares the object giving it a name "mycanvas" and a title, "A Set Of Histos":

```
TCanvas *mycanvas=new TCanvas("mycanvas","A Set Of Histos",1);
```

The third argument of the constructor '1', relates to the size of the TCanvas that is to be produced. By choosing '1' a default 700 × 500 pixel size is chosen. It is equally possible to manually decide the size of the overall TCanvas. For instance:

```
TCanvas *mycanvas=new TCanvas("mycanvas","A Set Of Histos",700,500);
```

is equivalent to the constructor used previously, the third and fourth arguments now take on the values of the width and height of the TCanvas in pixels. In ROOT classes can often be declared (or instantiated) by manyt different type of constructors. This takes advantage of the C++ feature of function overloading[2].

Once the object is instanitiated its member functions can be accessed via its pointer ('myclass'). The first one of these used is the 'Divide' function:

```
mycanvas->Divide(2,2);
```

[2]I.e. C++ will choose which constructor to use based on the number and types of arguments passed to it.

This simply splits the original canvas into 4 pads (2×2). These pads then become assigned a number individual to it from 0 to n-1, where n is the total number of pads. Before drawing a histogram, ROOT has to know to which pad it should be sent. For this reason the 'cd()' command is used.

```
mycanvas->cd(0);
```

This tells ROOT to make the pad assigbed the number '0' active. This active pad is where any subsequently drawn objects will end up. By changing the active pad by statements like this it is possible to draw each histogram in a separate pad as done in the example.

5.4 Applying Legends, labels, text and lines

ROOT by default does not label the axis of its histograms. But as you've probably guessed by now there are funcitons available to do it. The next example shows how this may be done as well as a few things you can do with a histogram, e.g. Legends and adding text, lines etc.

```
//Worked Example 5.3
//Labelling Axis, Legends, etc

#include<iostream>
#include "TH1D.h"
#include "TCanvas.h"
#include "TLegend.h"
#include "TLine.h"
#include "TPaveText.h"

int run(){
  //Create a histogram
  TH1D* histo1=new TH1D("myhisto1","myhisto",10,0,10);
  TH1D* histo2=new TH1D("myhisto2","myhisto2",10,0,10);

  //Fill the histogram
  histo1->Fill(3,4);
  histo2->Fill(8,7);

  //Label the axis
  histo1->SetXTitle("No. of squirrels");
  histo1->SetYTitle("frequency");

  //Example continues on next page
```

```
  //Reset the limits of the y-axis of the first histogram
  histo1->GetYaxis()->SetRangeUser(0,10);

  //Turn off statistics box
  histo1->SetStats(false);

  //Change the title of the histogram
  histo1->SetTitle("Squirrel Density in North-West Leicestershire");

  //Change the colour and line-style of the second histogram
  histo2->SetLineColor(2);
  histo2->SetLineStyle(2);
  //Create a Legend
  TLegend* leg=new TLegend(0.15,0.65,0.35,0.85);
  leg->AddEntry(histo1,"Loughborough","l");
  leg->AddEntry(histo2,"Kegworth","l");

  //Add a line
  TLine* line=new TLine(0,5,10,5);

  //Add some text
  TPaveText* ptext = new TPaveText(0.55,0.65,0.75,0.85,"NDC");
  TText *t1=ptext->AddText("Data from 1998");
  ptext->SetFillStyle(4000);
  ptext->SetBorderSize(0);

  //Draw the histograms
  histo1->Draw();
  histo2->Draw("same");

  //Draw the legend
  leg->Draw();

  //Draw the line
  line->Draw();

  //Draw the text
  ptext->Draw();

  return 0;
}
```

This should lead to the result as shown in Figure 5.2. The first adjustment is to add labels to the x and y axis by use of the SetXTitle and SetYTitle functions of the histogram class. The statistics box that appears in the top right corner of histograms by default can also be turned off by the SetStats(false) function. It is also possible to reset the title of a hisotgram after it has been instantiated by use of SetTitle(). Both the line style and its

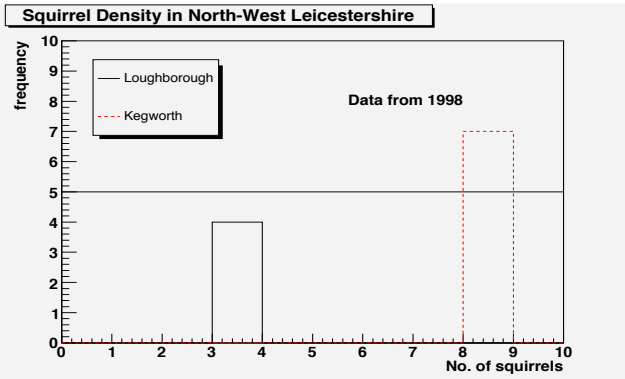

Figure 5.2: Histogram created by worked example 5.3.

colour can be altered (as in the example) and the argument (a number) chooses between different options.

In addition to the functions that modify the histogram's appearance it is also possible to add other objects to the canvas including a legend, a line and text. The legend makes use of the TLegend class (see relevent include statement '#include "TLegend.h"') and is instantiated by:

```
TLegend* leg=new TLegend(0.15,0.65,0.35,0.85);
```

This is a bit of an odd constructor. The four numerical arguments define the boundaries of the box which will contain the legend. The co-ordinate system in this case is referred to as 'NDC' (normalised co-ordinate system). This system is independent of the window's size and (0,0) represents the bottom left corner and (1,1) the top right[3]. The TLegend's constructor relates to two co-ordinate points which correspond to the bottom left and the top right corners of the TLegend box:

```
TLegend* leg= new TLegend(x1,y1,x2,y2);
```

Where 1 and 2 denote the different co-ordinate points. Once the TLegend object has been initiated different series may be added to it by use of the 'AddEntry' function:

```
leg->AddEntry(histo1,"Loughborough","l");
```

The TLegend object has to know which histogram object the key relates to and this is done by supplying the pointer to the relevant histogram (i.e. histo1 in this case) as the first argument. The second argument is the key you wish to apply (in this case "Loughborough"[4]) and the third is a style description "l". The "l" denotes that the key draws a line (of the style of the histogram in question) to denote the series. Other options "P" and "F" can be used to instead use markers or fill attributes instead (see the ROOT manual for details).

It is also possible to add a line using the TLine class (note '#include "TLine.h"'). Just to confuse matters the co-ordinates in this case relate to that of the histogram (i.e. the x,y values as defined by the axis) as opposed to the NDC used for the TLegend. The constructor simply gives the co-ordinates of the beginning and end of this line:

[3] i.e. the x and y co-ordinates both go from $0 \rightarrow 1$.

[4] http://en.wikipedia.org/wiki/Loughborough.

```
TLine* line=new TLine(x1,y1,x2,y2);
```

Where 1 and 2 denote the co-ordinates of the beginning and end of the line.

Adding text is a little more complicated. First a box has to be defined to hold it using a TPaveText object:

```
TPaveText* ptext = new TPaveText(0.55,0.65,0.75,0.85,"NDC");
```

This constructor is similar to that of the TLegend. However the final argument "NDC" specifies the co-ordinate system as NDC. If this is omitted then the co-ordinate system is taken to be that of the histogram (as in the case of the TLine). After the box has been defined text can be added

```
ptext->AddText("Data from 1998");
```

It is also possible to change the fill-style and border of the box. Here we set he border size ('SetBorderSize') to zero so it doesn't appear in the histogram.

5.5 Output Of Canvas to .eps, .jpeg ...

Once you have a TCanvas that is to your liking, you will probably want to export into a format that you can incorporate into Powerpoint presentations, latex files etc. This can be done in two ways.

The first is interactive. When a TCanvas window is open you wil notice there is a tool bar with the headings "File, Edit, View etc..". If you simply use the mouse to click on $File \rightarrow Save$ you will see a window appear with options for saving the TCanvas as a .ps, .eps, .pdf, .jpg , .gif etc.

The alternative is to do it in the script using the TCanvas's 'Print' function this is shown in the worked example below.

```
//Worked Example 5.4
//Exporting a Canvas to a .ps, .eps, .jpg file.

#include<iostream>
#include "TH1D.h"
#include "TCanvas.h"

int run(){
  //Create a histogram
  TH1D* histo=new TH1D("myhisto","myhisto",10,0,10);

  //Fill the histogram
  histo->Fill(5);

  //Create a Canvas and draw the histogram
  TCanvas* mycanvas=new TCanvas("mycanv","mycanv",1);
  mycanvas->cd(1);
  histo->Draw();

  //Export Canvas to files
  mycanvas->Print("mycanv.ps");
  mycanvas->Print("mycanv.eps");
  mycanvas->Print("mycanv.gif");
  mycanvas->Print("mycanv.jpg");

  return 0;
}
```

In this example the TCanvas (with pointer mycanvas) is exported as four different files mycanv.ps, mycanv.eps, mycanv.gif and mycanv.jpg. This is done in the four Print() function calls at the end of the script e.g:

```
mycanvas->Print("mycanv.ps");
```

The argument of the Print function gives the name of the output file. ROOT is rather clever in that it interprets the name you give the output file and decides on what format the file should be. So if the extension to the file is '.ps' as in the case above then output file will be of PostScript form[5].

5.6 How to avoid that dull grey background

You will notice in all the histograms that I have placed in this note that there is a dull grey edge to all of them. This is the ROOT default and I left it there for simplicity's sake. However, it can be removed and (in my opinion) the histograms look better without this border. This is shown in the next example:

[5].eps=encapsulated PostScript, etc. you get the general idea....

```
//Worked Example 5.5
//How to avoid the dull grey background

#include<iostream>
#include "TH1D.h"
#include "TCanvas.h"

int run(){
  //Change Drawing style
  gROOT->ProcessLine("gStyle->SetCanvasColor(0)");

  //Create a histogram
  TH1D* histo=new TH1D("myhisto","myhisto",10,0,10);

  //Fill the histogram
  histo->Fill(5);

  //Create a Canvas and draw the histogram
  TCanvas* mycanvas=new TCanvas("mycanv","mycanv",1);
  mycanvas->cd(1);
  histo->Draw();

  //Export Canvas to files
  mycanvas->Print("mycanv.eps");

  return 0;
}
```

The result is shown in Figure 5.3. The command which removes the grey background is:

```
gROOT->ProcessLine("gStyle->SetCanvasColor(0)");
```

This has a rather odd-structure. What we're actually doing here is calling the CINT interpreter from within our compiled script in order to set the canvas style. Finer control of the style can be made with similar commands, but we won't go into them here and you can see the ROOT manual if you want more details.

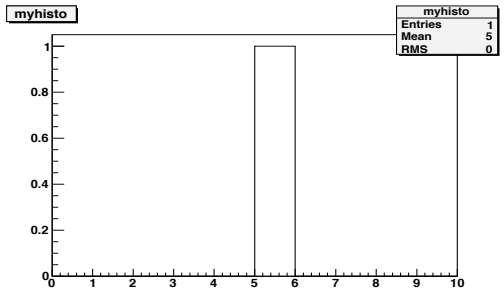

Figure 5.3: Histogram created by worked example 5.5. Note the absence of the dull grey border.

5.7 Putting Formulae Into Histogram Titles

You might want to label the axis of your histogram with formulae (e.g. α_s etc). ROOT allows you to do this by use of 'LATEX' like commands. In the following example this is shown for a simple histogram:

```
//Worked Example 5.6
//Putting Formulae/Symbols into Histogram Titles

#include<iostream>
#include "TH1D.h"
#include "TLatex.h"

using namespace std;

int run(){
  //Create a histogram
  TH1D* myhisto=new TH1D("myhisto","myhisto",10,0,10);

  //Fill it
  myhisto->Fill(4);

  //Label the Axis
  myhisto->SetXTitle("Squirrel Terrorism Quotient #alpha^{#mu}");
  myhisto->SetYTitle("#frac{y_{n}+#Delta}{y_{n}-#Delta}");

  //Draw the histogram
  myhisto->Draw();

  return 0;
}
```

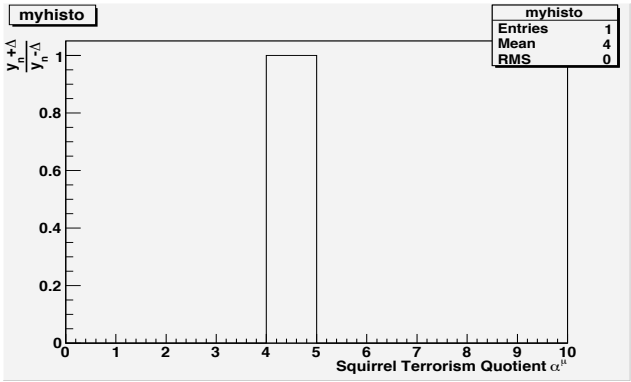

Figure 5.4: Histogram created by worked example 5.6.

This should produce a histogram as shown in Figure 5.4. The labelling of the axis is done in the lines:

```
myhisto->SetXTitle("Squirrel Terrorism Quotient #alpha^{#mu}");
myhisto->SetYTitle("#frac{y_{n}+#Delta}{y_{n}-#Delta}");
```

The '#' character denotes that a LATEX like formula interpretation is to be used. As can bes seen '#alpha' creates the symbol α and similarly for the other Greek letters. Furthermore, you are able to use the LATEX superscript and subscript operators in their usual manner. Some other options are available such as producing fractions '#frac{}{}'. For a full description of the LATEX functionality within ROOT see the official manual[6].

[6]I think the best description is given within the chapter on the 'Graphics and the graphical user interface'.

Chapter 6

Handling lots of objects

6.1 Arrays and Vectors

We showed in Chapter 3, that it was possible to relatively easily create an array to store histogram objects[1]. However, the same could have been done with vectors. Vectors can be considered as a C++ style array. They have advantages over normal arrays as they carry information on their size (i.e. the number of entries) with them and do not have to be defined to be a certain size at the outset.

The first worked example is a mirror of example 3.7 except using a vector. Here we create a 100 histograms, store them in a vector and then draw them:

```
//Worked Example 6.1
//Creating one hundred histograms in a vector

#include<iostream>
#include<sstream>
#include<vector>
#include "TFile.h"
#include "TH1D.h"
#include "TString.h"

using namespace std;

//A function to convert an integer to a TString
TString ToString(int num) {
  ostringstream start;
  start<<num;
  TString start1=start.str();
  return start1;
}

//Example continues on next page
```

[1] Or more technically speaking their pointers.

```
int run(){

  //Declare a vector of histogram pointers
  vector<TH1D*> histos;

  //Start a loop and create the histograms
  for(int i=0; i<100; i++){

    //Convert the integer 'i' to a TString
    TString numstr=ToString(i);

    //Create the histogram's name
    TString histoname="histo"+numstr;

    //Declare the histogram
    TH1D* hist=new TH1D(histoname,histoname,100,0,100);
    hist->Fill(i);

    //Put the histogram pointer into a vector
    histos.push_back(hist);
  }


  //Loop over all the vector entries
  for(int i=0; i<histos.size(); i++){
    if(i==0){
      (histos[i])->Draw();
    }
    else{
      (histos[i])->Draw("same");
    }
  }

  return 0;
}
```

This should produce the same output as seen in Figure 3.2. The C++ vector is an object and needs the appropriate '#include¡vector¿' line in the preamble. The vector is instantiated using the C++ syntax to create a container of TH1D pointers. Note that the actual size of the vector is not defined at initiation.

```
vector<TH1D*> histos;
```

The vector can be treated like a stack with elements being progressively added using the 'push_back' command e.g:

```
histos.push_back(hist);
```

Furthermore the number of array entries can be found using the size() member function:

```
histos.size()
```

The actual members of the vector can be found in the normal manner for arrays by use of the subscript operator []. It is beyond the scope of this note to go into detail on vectors, which are really a C++ (rather than specifically ROOT) thing. However, I hope you can see from this example that there might be some situations where using them might be worth the extra hassle.

6.2 Lists

Another option for storing lots of ROOT objects is to use a TList. A TList is like a vector in that the size of the list does not have to be defined at instantiation. There are however further advantages; a TList can access objects based on their name and store a variety of object types in a single structure. A couple of examples of using a TList are given in what follows. We will again create 100 histograms and this time store them to a list from which they will be drawn and saved to a file.

```
//Worked Example 6.2
//Creating one hundred histograms in a TList

#include<iostream>
#include<sstream>
#include "TList.h"
#include "TFile.h"
#include "TH1D.h"
#include "TString.h"

using namespace std;

//A function to convert an integer to a TString
TString ToString(int num) {
  ostringstream start;
  start<<num;
  TString start1=start.str();
  return start1;
}

//Example continues on next page
```

```
int run(){
  //Declare a TList
  TList* list=new TList();

  //Start a loop and create the histograms
  for(int i=0; i<100; i++){

    //Convert the integer 'i' to a TString
    TString numstr=ToString(i);

    //Create the histogram's name
    TString histoname="histo"+numstr;

    //Declare the histogram
    TH1D* hist=new TH1D(histoname,histoname,100,0,100);
    hist->Fill(i);

    //Put the histogram pointer into the list
    list->Add(hist);
  }

  //Loop over the list entries
  TH1D *source = (TH1D*)list->First();

  while(source){
    if(source==(list->First())){
      source->Draw();
    }
    else {
      source->Draw("same");
    }

  source=(TH1D*)(list->After(source));
  }

  //Record all the list entries to a file
  TFile f1("outwex6.2.root","RECREATE");
  list->Write();

  return 0;
}
```

The TList object as always needs an include statement if you want to use it (#include "TList.h"). Declaring a TList object is simple:

```
TList* list=new TList();
```

You will notice that you do not have to specify either the size of the list or the type of

the objects that are going to be stored. The reason we can get away with this comes from the inheritance structure of the ROOT C++ classes. The more complicated classes such as histograms etc, are all derived from a simple base class called a TObject. Because of this common base class a container such as a TList can just store this basic part and we can return to the complex (derived) object by type-casting as we will see shortly.

The histograms are added to the TList simply by doing:

```
list->Add(hist);
```

Where 'hist' is the histogram pointer. In reality it is only the TObject component of the histogram (TH1D) class that is stored. After all the objects pointers have been stored we can loop over the list members. There are a number of ways of doing this but one of the easiest is to step through each, one at a time. In a TList the elements form a chain, e.g if there are 3 entries the list remembers that 2 follows 1 and 3 follows 2 etc. This is complemented with a knowledge of the first and last members of the list.

Looping through the TList is thus achieved by finding the first element of the list via:

```
TH1D *source = (TH1D*)list->First();
```

Then cycling through consecutive members by finding the object which is after this one:

```
source=(TH1D*)(list->After(source));
```

You will notice that it is necessary to cast the result of list→First() and list→After() as TH1D pointers. The reason for this is that the TList only remembers that the TObject part of the objects it contains and not the full complex histogram. We have to tell ROOT what the object is before we can use it.

Recording results from a TList to a file is very easy and a simple Write() command ensures that all objects in the list are recorded.

```
list->Write();
```

The TList gains in flexibility with the cost of a slightly more complicated syntax for extracting objects. There is however a further advantage of TLists in being able to extract objects using their name.

You will recall that all ROOT objects (that inherit from TObject) require a name[2] (e.g. the first argument of a TH1D constructor.). A TList has an option to obtain an object based only on its name. This is shown in the next example. Here the file of histograms is read in and initialised in a TList. Following this an individual histogram is picked by name and drawn.

[2]This is distinct from the title.

```
//Worked Example 6.3
//Obtaining an object from a TList by name

#include<iostream>
#include "TH1D.h"
#include "TList.h"
#include "TFile.h"
#include "TKey.h"

//Function to collect all the objects in a TFile into a TList

TList* GetFileObjects(TFile* f1){
  //Create a list
  TList* list=new TList();

  //Loop over file contents

  //Create an iterator
  TIter nextkey( f1->GetListOfKeys() );
  TKey *key;

  //Loop over the keys in the file
  while ( (key = (TKey*)nextkey())) {

    // Read object from first source file
    TObject *obj = key->ReadObj();

    //Cast the TObject pointer to a histogram one.
    TH1* histo = (TH1*)(obj);

    //Take ownership of the histogram object
    histo->SetDirectory(0);

    //Record the histogram to a TList
    list->Add(histo);
  }

  return list;
}

//Example continues on next page
```

```
int run(){
  //Open up the file
  TFile* f1=new TFile("outwex6.2.root");

  //Get a list of all TObjects in the file
  TList* list=GetFileObjects(f1);

  //Find the object in the list with name "histo23"
  TH1D* histo=(TH1D*)(list->FindObject("histo23"));

  //Draw this histogram
  histo->Draw();

  //Clean up
  delete f1;

  return 0;
}
```

Here, the program is split into two functions[3], GetFileObjects and run(). The GetFileObjects() function takes the file given as an argument and loops over its entries adding them to the TList which it then returns. This looping is identical to used in worked example 3.8.

As you will recall, when the file was made the histograms were given names: histo1, histo2 etc. These can be used to extract the object you want using the FindObject() function of a TList. This is done in the run() function:

```
TH1D* histo=(TH1D*)(list->FindObject("histo23"));
```

This finds the object in the list called "histo23". As a TList only considers TObjects this has to be type-cast into a TH1D pointer, hence the (TH1D*). Once this is done the pointer 'histo' can be used to draw the histogram as normal. The result should be as shown in Figure 6.1.

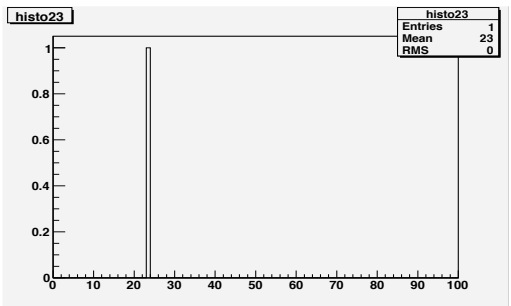

Figure 6.1: Histogram created by worked example 6.3

[3]As run() is the controlling main() type function this script can again be executed like the other examples.

If a systematic object (histogram) naming process is adopted then a TList can be a very good way of organising lots of histograms. This can be made even more useful by use of the function we saw earlier which converts an integer to a TString.

Chapter 7

Different Script Types

7.1 Why Script Type Matters

As wan described in the opening chapter, there are many ways of running root. These include:

1. Via the CINT interactive interpreter.
2. Using a semi-compiled script with ACLiC.
3. From C++ linking the ROOT libraries externally.

In this note, we have been using method 2, with the ACLiC libraries and compiling our scripts with lines like:

```
root [0] .L wex1.C++
```

All of the examples given would have worked equally well with method 1 (the CINT interpreter). The only difference for running the examples without compiling is that the '++' is omitted when loading the script e.g:

```
root [0] .L wex1.C
```

When running under CINT there are advantages in that it is sometimes possible to ignore include statements and on occassions it is possible to do things that aren't possible in compiled C (for instance interchanging pointers and objects). I personally would suggest compiling though as it makes debugging code much easier and the resultant code can be executed approximately 10 times faster.

7.2 The Problem With CINT

The following worked example doesn't work, but this is intentional.

```
//Worked Example 7.1
//Problems with CINT

#include<iostream>
#include<sstream>
#include "TList.h"
#include "TFile.h"
#include "TH1D.h"
#include "TString.h"

using namespace std;

//A function to convert an integer to a TString
TString ToString(int num) {
  ostringstream start;
  start<<num;
  TString start1=start.str();
  return start1;
}

int run(){
  //Create an array of histograms
  TH1D* histoarray[10];

  for(int i=0; i<10; i++){
    TString num=ToString(i);
    TString name="histo"+num;
    histoarray[i]=new TH1D(name,name,20,0,20);
    histoarray[i]->Fill(i+1);
  }

  //This shouldn't work as 'i' is nor defined
  histoarray[i]->Draw();

  return 0;
}
```

The problem with the code is in the line:

```
histoarray[i]->Draw();
```

Outside the 'for' loop the 'i' variable has no meaning and hence there is no way for this to work. If you use CINT (i.e. .L wex7.1.C without ++ on end) there are no errors until the script is executed:

```
root [0] .L wex7.1.C
root [1] run()
Error: illegal pointer to class object histoarray[i] 0x0 360
FILE:wex7.1.C LINE:34
*** Interpreter error recovered ***
root [2]
```

The error report from CINT in this case is useful as it describes something wrong with a pointer on line 34 (which in my code is the offending line). However this error report isn't as full as that received when using ACLIC e.g:

```
root [0] .L wex7.1.C++
Info in <TUnixSystem::ACLiC>: creating shared library
/home/clements/rootmanual/code/ch7/./wex7.1_C.so
In file included from /home/clements/rootmanual/code/ch7/fileAmSWcp.h:32,
                 from /home/clements/rootmanual/code/ch7/fileAmSWcp.cxx:16:
/home/clements/rootmanual/code/ch7/wex7.1.C: In function `int run()':
/home/clements/rootmanual/code/ch7/wex7.1.C:33: name lookup of `i' changed for
   new ISO `for' scoping
/home/clements/rootmanual/code/ch7/wex7.1.C:25:   using obsolete binding at `i'
g++: /home/clements/rootmanual/code/ch7/./fileAmSWcp.o:
No such file or directory
Error in <ACLiC>: Compilation failed!
root [1]
```

Firstly the code fails to compile, and will not run. Secondly the error messages are far more lucid. It tells us the problem is in function run(), that the problem is associated with the lookup of the 'i' variable as well as the problem line number 33 (with 25 being given as the original definition of 'i'). Identifying problems in code is tricky and the more information you can get from your compiler the better.

For me this is the main reason for choosing to compile your script. When things go wrong (and they will) it's much easier to debug. There is also an added bonus of faster execution time when the CINT interpreter is not used. This can be as much as a factor of 10 for ntuple/tree analysis.

7.3 Fully Compiled C++ with ROOT

Another option is to fully compile the code as external C++ and gain ROOT functionality by including the relevant libraries. This in a way guarantees that the basis you are working upon can enjoy all the benefits of C++. A worked example of this is given below:

```
//Worked Example 7.2
//Compiling C++ with ROOT external

#include<iostream>
#include "TH1D.h"
#include "TFile.h"

using namespace std;

int main(){
  //Create a histogram
  TH1D* myhisto=new TH1D("myhisto","myhisto",10,0,10);

  //Fill the histogram
  myhisto->Fill(3);

  //Save the histogram
  TFile f1("outwex7.2.root","RECREATE");
  myhisto->Write();

  cout<<"Program finished"<<endl;

  return 0;
}
```

Notice that as we're now using C++ properly we have to have a function called main() in our program. To compile this properly we need to link the ROOT libraries, this is done in the Makefile:

```
CC = g++
FLAGS = -O3 -fpic
ROOTINCS=`root-config --cflags`
ROOTLIBS=`root-config --libs`


wex7.2: wex7.2.C
$(CC) -o wex7.2.exe ${FLAGS} wex7.2.C ${ROOTINCS} ${ROOTLIBS}
```

To run the program at the Linux command line do:

```
[clements@ppepc10 ch7]> make wex7.2
g++ -o wex7.2.exe -O3 -fpic wex7.2.C `root-config --cflags`
`root-config --libs`

[clements@ppepc10 ch7]> ./wex7.2.exe
Program finished
```

This should produce a file "outwex7.2.root" with a saved histogram in the same directory. There are clearly benefits to running your code in this way, however it becomes tricky to draw histograms[1]. It is entirely possible though to do an analysis, save data and then look at the results interactively afterwards and indeed this is often a good idea.

[1] I have yet to find out how to make this work.

Chapter 8

Fitting Histograms and TMinuit

8.1 Fitting a Histogram

ROOT has a wide range of tools for fitting histograms with analytic functions after they have been created. There are a variety of methods of going about this: via the graphical interface and a few different conventions for scripts. Indeed the ROOT manual has an entire chapter dedicated to this subject.

In the following example I use a 'user-defined' function to fit a 1-D histogram and how to extract the fit parameters. For simplicity the 'user-defined' function is simply a straight line in this case. The fitting mechanism by default draws the histogram and the fitted line which should look like Figure 8.1.

```
//Worked Example 8.1
//Fitting a Straight Line To A Histogram

#include <iostream>
#include "TH1D.h"
#include <TMath.h>
#include <TROOT.h>
#include "TF1.h"

using namespace std;

//Declare a user defined function for fitting
Double_t fitf(Double_t *x,Double_t *par)
{
  Double_t value=par[0]+(par[1]*x[0]);
  return value;
}

int run(){
  //Create a histogram to be fitted
  TH1D* myhisto=new TH1D("myhisto","myhisto",10,0,10);

  //Fill the histogram
  for(int i=1; i<=10; i++){
    myhisto->SetBinContent(i,i+2);
  }

  //Create a TF1 object with the function as defined above...
  //The last three parameters specify the range and number of
  //parameters of the function.
  TF1 *func=new TF1("fit",fitf,0,10,2);

  //Set the initial parameters of the function
    func->SetParameters(0.14,0.7);

    //Give the parameters meaningful names
    func->SetParNames("intercept","gradient");

    //Call TH1::Fit with the name of the TF1 object
    myhisto->Fit("fit");

    //Access the Fit results directly
    cout<<"The y-intercept is:"<<func->GetParameter(0)<<endl;
    cout<<"The gradient is:"<<func->GetParameter(1)<<endl;

    cout<<"Fitting completed....."<<endl;

  return 0;
}
```

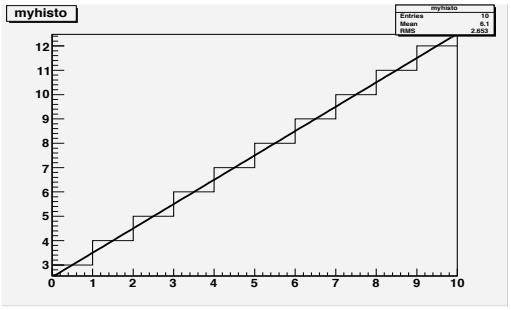

Figure 8.1: Histogram created by worked example 8.1

The example is split into two functions fitf() and run()[1]. The 'fitf()' is the user-defined function that ROOT uses to fit the histogram with. For the fitting to work it must have the exact form:

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

I.e. be called 'fitf' and take the arguments of a pointer to a 'Double_t' called 'x' and 'par'. These pointers can be considered as arrays in C++ relating to the dimensions of the 'histogram' being fitted (x) and the fit parameters used in the functional form of the fit. In other words, if you have a 1-D histogram that you want to fit, then x[0] relates to the histograms x-axis. If you have a 2-D histogram then x[0] relates to the x-axis and x[1] to the y-axis. The parameters are denoted by par[0], par[1] etc.

In our example above the 'user-defined' function has a value:

```
Double_t value=par[0]+(par[1]*x[0]);
```

This is where our straight-line fit is defined. X[0] relates to the x-variable of the histogram, par[0] is the y-intercept and par[1] is the gradient of the straight line.

In the 'run()' function a histogram is first declared and filled. Then begins the process of fitting this histogram. Firstly a ROOT 'TF1' function object[2] has to be delcared. As this is another ROOT object we need to have the appropriate include statement inthe preamble '#include "TF1.h"'. The TF1 objects is constructed thus:

```
TF1 *func=new TF1("fit",fitf,0,10,2);
```

The first argument 'fit' is the name of the ROOT object (remember all ROOT objects must have a unique name. The second argument relates to the fitting function we defined in 'fitf'[3]. The final three arguments relate to the minimum and maximum x-values of the function (0,10) and the number of parameters that define it (in this case 2 (intercept and gradient).

After the basic form of the function has been declared, initial values of the parameters have to be given before the fitting process can be done this is done in the line:

```
func->SetParameters(0.14,0.7);
```

[1]The script is executed via the run() function in the normal manner

[2]The TF1 class is documented here: http://root.cern.ch/root/html/TF1.html

[3]N.B. To use this method of fitting this argument must always be 'fitf'

This sets up the parameters par[0] (the intercept) and par[1] the gradient (as defined in the function fitf() to the values 0.14 and 0.7 respectively. The idea is to provide approximate values of the parameters you are trying to find in order to give the fitting process a fighting chance. We are also able to assign a name to the parameters to make reading the results a little easier:

```
func->SetParNames("intercept","gradient");
```

This sets the names of the parameters par[0] and par[1] to "intercept" and "gradient" respectively. Both the SetParameters() amd SetParNames() functions can be extended in the event of more parameters by adding more arguments e.g:

```
func->SetParNames("name1","name2","name3","name4",...);
```

This can be done for a maximum of 10 names (or values). In the event that you have more you must use an alternative function which sets names individually e.g:

```
func->SetParName(11,"squirrel-correction");
```

This gives the 11th parameter of a function 'func' the name "squirrel-correction". The first argument is the number of the parameter, the second is the name. A similar approach can be used for parameters: func-¿SetParameter(Int_t ipar, Double_t parvalue).

After all this setting up the 'Fit()' function can be called. This is a member function of the histogram class (TH1D) and so its pointer is used to call it:

```
myhisto->Fit("fit");
```

The name of the function "fit" has to be used to identify which function is being used to do the fitting. Recall the name was first used in the TF1 constructor. The 'Fit()' function by default draws both the histogram and the fitted-function. In addition you will notice that it provides some information on how the fitting went at the ROOT command line:

```
FCN=9.3829e-25 FROM MIGRAD    STATUS=CONVERGED    30 CALLS          31 TOTAL
                    EDM=1.8753e-24    STRATEGY= 1      ERROR MATRIX ACCURATE
 EXT PARAMETER                                          STEP         FIRST
 NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
  1  intercept    2.50000e+00   1.31536e+00   3.85633e-04   2.04481e-12
  2  gradient     1.00000e-00   2.81437e-01   8.25109e-05   3.84444e-12
```

This table provides the value and predicted error of the fitted values (gradient and intercept) along with some information about how many iterations were required. You can access the values of a fitted parameters directly by:

```
func->GetParameter(ipar)
```

Where 'ipar' is the parameter number i.e 0 relates to par[0] etc. The predicted errors on the fit values can be obtained similarly through func-¿GetParError(ipar).

The 'user-defined' fitting procedure offers a great deal of flexibility, but at the cost of less than intuitive interface. Its perhaps best to used a worked example like this (or another) as a recipe and work from there.

8.2 Built-in Maths Functions in ROOT

ROOT has a selection of built-in maths functions which can sometimes make life easier. These can be found at http://root.cern.ch/root/html/TMath.html and include functions for gaussians, Breit-Wigner, Landau etc. The next worked example shows how you can use on of these directly to get a TF1 function (n.b. we're not doing any fitting here).

```
//Worked Example 8.2
//Using the built-in maths functions

#include<iostream>
#include "TF1.h"
#include "TMath.h"

using namespace std;

int run(){

  //Create a Gaussian TF1 object
  TF1* func1=new TF1("func1","TMath::Gaus(x,[0],[1])",0,10);
  func1->SetParameter(0,5);
  func1->SetParameter(1,2);

  //Integrate the function
  cout<<"The integral of the function is:"<<func1->Integral(0,10)<<endl;
  cout<<"The mean of the function is:"<<func1->Mean(0,10)<<endl;
  cout<<"The variance of the function is:"<<func1->Variance(0,10)<<endl;

  //Draw the function
  func1->Draw("alp");

  return 0;
}
```

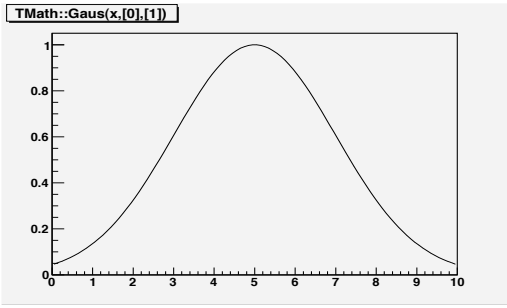

Figure 8.2: Histogram created by worked example 8.2

The result should look like that in Figure 8.2. Again the TF1 class is used but this time it is declared in a different way:

```
TF1* func1=new TF1("func1","TMath::Gaus(x,[0],[1])",0,10);
```

The first argument 'func1' is again the name of the object. The second defines the function form "TMath::Gaus(x,[0],[1])". This calls the built in Gaussian function (Gaus) for the variable 'x' with the mean and variance determined bu the first and second parameters (denoted by [0] and [1] respectively). The final two arguments determin the range (in x of the function). The parameters have to be set outside the constructor call by doing:

```
func1->SetParameter(0,5);
func1->SetParameter(1,2);
```

Which sets the first parameter [0] to the values of 5 and the second parameter [1] to the value of 2. The "TMath::" has to be prepended to the relevant TMath function so that ROOT knows where to look to find the definition.

After the function has been set up there are a variety of options for caluculatig integrals, means and variances defined within boundaries. The resulting function can also be drawn:

```
func1->Draw("alp");
```

We have to add the arguments "alp" to the Draw command for TF1 objects (and graphs). This tells ROOT to draw an axis (a), to connect points with a line (l) and to add point markers (p). For some reason such arguments are not needed as a default in the case of histograms (i.e. TH1D objects).

Its worth taking a note of the built in functions as they can save time when you are doing fits and offer nice features such as performing numerical integrations etc. You can also get ROOT to provide random numbers distributed according to a TF1's functional form using the GetRandom() function which can be handy for toy Monte-Carlo programs.

8.3 Using TMinuit

ROOT has its own version of the famous Fortran 'Minuit' program. For those unfamiliar with Minuit it is a numerical method which attempts to find the global minimum of a function in a multi-dimensional space. There is no formal mathematicall method of determining such a minimum, however, Minuit attempts to find it by doing clever things.

If anything the TMinuit interface is more arcane than that used to fit histograms, however, the following example should illustrate the key features:

```
//Worked Example 8.3
//An example of using TMinuit

#include <iostream>
#include <TMinuit.h>
#include <TApplication.h>
#include "TF1.h"
#include "TF2.h"
#include "TMath.h"

//Globally defined function
TF2 *f2 = new TF2("f2","(x*x)+(y*y)",-10,10,-10,10);

//The function to be minimised
void fcn(int& npar, double* deriv, double& f, double par[], int flag){
  f=f2->Eval(par[0],par[1]);
}

int run(){
  //First Draw function
  f2->Draw("LEGO2");
  cout<<"Program finished.."<<endl;

  //Setup the minimisation procedure
  const int npar = 2;               // the number of parameters
  TMinuit minuit(npar);
  minuit.SetFCN(fcn);

  double par[npar];                // the start values
  double stepSize[npar];           // step sizes
  double minVal[npar];             // minimum bound on parameter
  double maxVal[npar];             // maximum bound on parameter
  string parName[npar];

  par[0] = 7.0;      // a guess
  stepSize[0] = 0.1; // take e.g. 0.1 of start value
  minVal[0] = -10;   // if min and max values = 0, parameter is unbounded.
  maxVal[0] = 10;
  parName[0] = "x";

  par[1] = 7.0;      // a guess
  stepSize[1] = 0.1; // take e.g. 0.1 of start value
  minVal[1] = -10;   // if min and max values = 0, parameter is unbounded.
  maxVal[1] = 10;
  parName[1] = "y";
```

```
  //Setup parameters
  for (int i=0; i<npar; i++){
    minuit.DefineParameter(i, parName[i].c_str(),
   par[i], stepSize[i], minVal[i], maxVal[i]);
  }

  cout<<"Running Migrad()...."<<endl;
  // Do the minimization!
  minuit.Migrad();       // Minuit's best minimization algorithm
  cout<<"Migrad() completed..."<<endl;

  //Get the Minuit results
  double outpar[npar], err[npar];
  for (int i=0; i<npar; i++){
      minuit.GetParameter(i,outpar[i],err[i]);
      cout<<"Fitted parameter:"<<i<<" is:"<<outpar[i]<<" +/- "<<err[i]<<endl;
  }

  cout<<"Program Finished"<<endl;

 return 0;
}
```

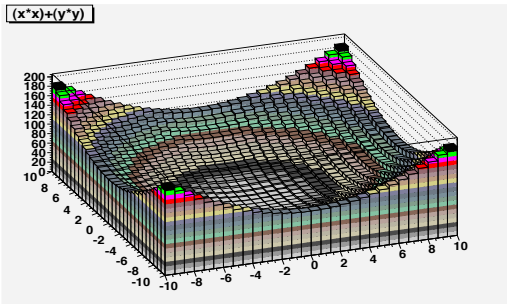

Figure 8.3: Histogram created by worked example 8.3

The program defines a two-dimensional function $x^2 + y^2$ which has a minimum at the origin (0,0) as shown in the Figure 8.3. Clearly in this case we wouldn't need Minuit to find the global minimum, but it serves as a good example. The function is set up as a TF2 object (http://root.cern.ch/root/html/TF2.html) within the boundaries of $< -10 < x, y < 10$.

The TMinuit system requires a function called 'fcn' to be defined with the exact structure:

```
void fcn(int& npar, double* deriv, double& f, double par[], int flag)
```

This is the function to be minimised. When TMinuit is invoked it uses the 'fcn()' function and varies the parameters par[][4] to arrive at the minimum value of 'f'. The

[4]npar is the number of parameters

user definition of 'fcn()' therefore must relate the parameters (par[]) to the function to be minimised. In this example this is done by evaluating the previously TF2 function at point $x = par[0]$, $y = par[1]$ in fnc():

```
f=f2->Eval(par[0],par[1]);
```

In the run() function, first the TF2 function is drawn and then Migrad (the TMinuit program) is set up first the number of parameters and the function it uses are declared (n.b. this must be called fcn):

```
const int npar = 2;       // the number of parameters
TMinuit minuit(npar);
minuit.SetFCN(fcn);
```

After this the starting values of the parameters, a step size and any boundaries are declared and intitialised through:

```
minuit.DefineParameter(i, parName[i].c_str(),
                       par[i], stepSize[i], minVal[i], maxVal[i]);
```

Where 'i' is the index of the parameter 0,1,2 etc, followed by name, starting values, step-size[5] and boundaries. After this TMinuit is called simply by:

```
minuit.Migrad();
```

TMinuit prints results to the command-line relating to the fit-paramters and their quality:

[5]The step-size is a rough measure of how far TMinuit should look around the initial parameters for each iteration.

```
 PARAMETER DEFINITIONS:
    NO.   NAME         VALUE      STEP SIZE      LIMITS
     1 x            7.00000e+00  1.00000e-01   -1.00000e+01  1.00000e+01
     2 y            7.00000e+00  1.00000e-01   -1.00000e+01  1.00000e+01
Running Migrad()....
 **********
 **    1 **MIGRAD
 **********
 FIRST CALL TO USER FUNCTION AT NEW START POINT, WITH IFLAG=4.
 START MIGRAD MINIMIZATION.  STRATEGY  1.  CONVERGENCE WHEN EDM .LT. 1.00e-04
 FCN=98 FROM MIGRAD    STATUS=INITIATE        8 CALLS           9 TOTAL
                     EDM= unknown      STRATEGY= 1      NO ERROR MATRIX
  EXT PARAMETER                CURRENT GUESS       STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  x            7.00000e+00   1.00000e-01   1.40046e-02   9.99799e+01
   2  y            7.00000e+00   1.00000e-01   1.40046e-02   9.99799e+01
 MIGRAD MINIMIZATION HAS CONVERGED.
 MIGRAD WILL VERIFY CONVERGENCE AND ERROR MATRIX.
 COVARIANCE MATRIX CALCULATED SUCCESSFULLY
 FCN=1.21007e-13 FROM MIGRAD    STATUS=CONVERGED      39 CALLS   40 TOTAL
                     EDM=2.42014e-13    STRATEGY= 1      ERROR MATRIX ACCURATE
  EXT PARAMETER                                        STEP         FIRST
  NO.   NAME      VALUE            ERROR          SIZE      DERIVATIVE
   1  x           -2.45974e-07   9.98334e-01   4.88307e-05  -4.91949e-06
   2  y           -2.45974e-07   9.98334e-01   4.88307e-05  -4.91949e-06
 EXTERNAL ERROR MATRIX.    NDIM=  25    NPAR=  2    ERR DEF=1
  1.000e+00 -1.138e-18
 -1.138e-18  1.000e+00
 PARAMETER  CORRELATION COEFFICIENTS
       NO.  GLOBAL      1      2
        1  0.00000   1.000 -0.000
        2  0.00000  -0.000  1.000
Migrad() completed...
```

The first part gives the initial value of the parameters (VALUE) and the intitial step-size (ERROR). If all goes well then TMinuit should say something along the lines of 'MIGRAD MINIMIZATION HAS CONVERGED' as above. The later parts give the value of the minimised fit paramters and their errors. In this case both x and y have a value of -2.45974×10^{-7} which is close to the value of 0 which we expect and certainly within the stated error.

We can get access to the parameters directly by doing:

```
minuit.GetParameter(i,outpar[i],err[i]);
```

Which for parameter 'i' sets array values outpar[i] and err[i] to its value and error.

Again it is probably best if using TMinuit to use this example (or another) as a template and work from there.