Outline 2017/04/21

- Classes as data containers
- TH2, TProfile
- TGraph
- Collection classes (TList, TClonesArray)
- Exercises

Classes as data containers

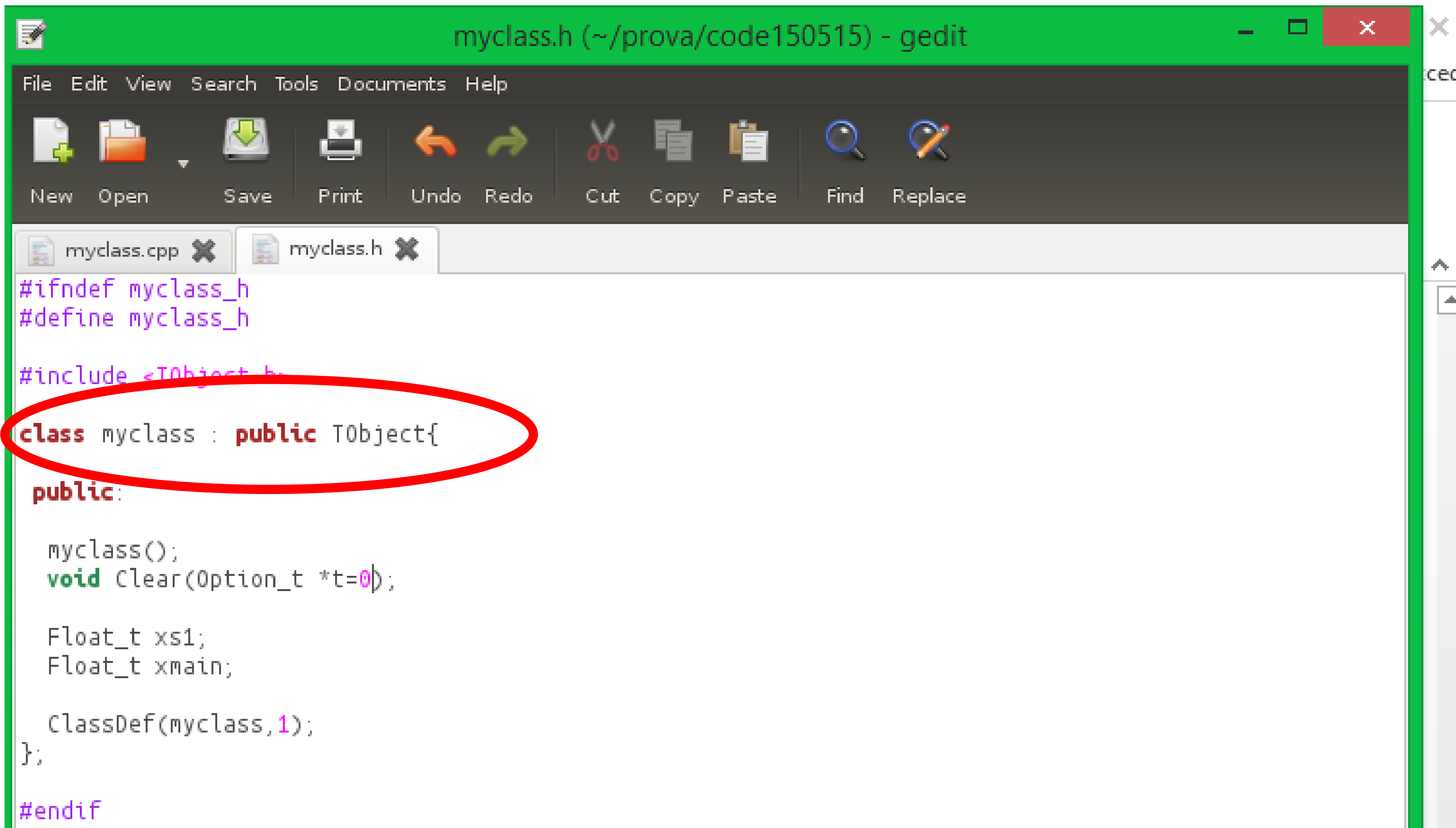
Classes used as data containers:

1. must have all data-variables declared as public;
2. can have any other method, variable, etc. declared any other way (public, protected or private).

Convenience:

1. methods handling variables can be included;
2. easier way to declare many variables when creating and reading the TTree.

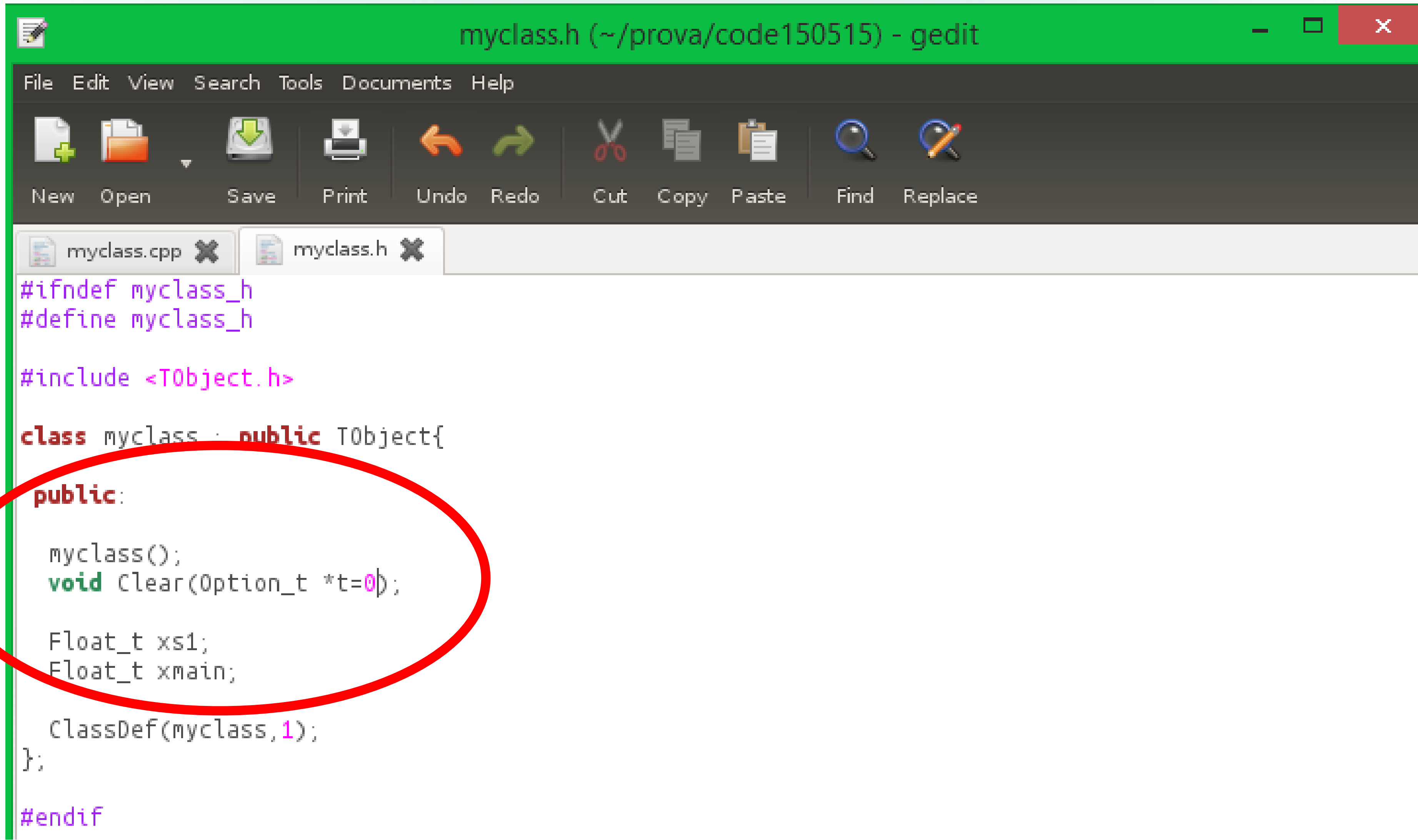
Classes as data containers, example




```
#ifndef myclass_h
#define myclass_h

#include <TObject.h>

class myclass : public TObject{

 public:

  myclass();
  void Clear(Option_t *t=0);

  Float_t xs1;
  Float_t xmain;

  ClassDef(myclass,1);
};

#endif
```

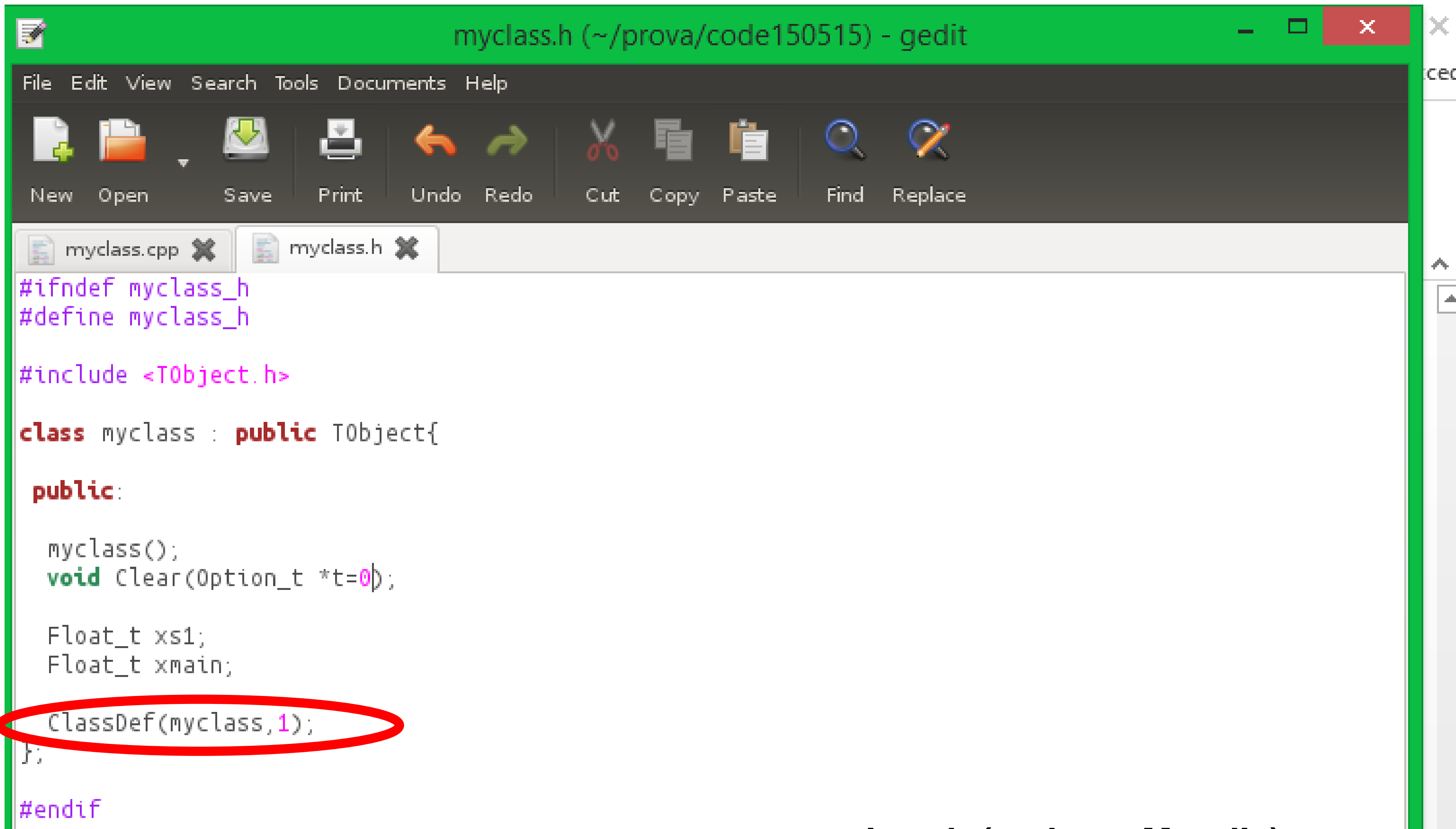
myclass.h (code on Moodle)

Classes as data containers, example



myclass.h (code on Moodle)

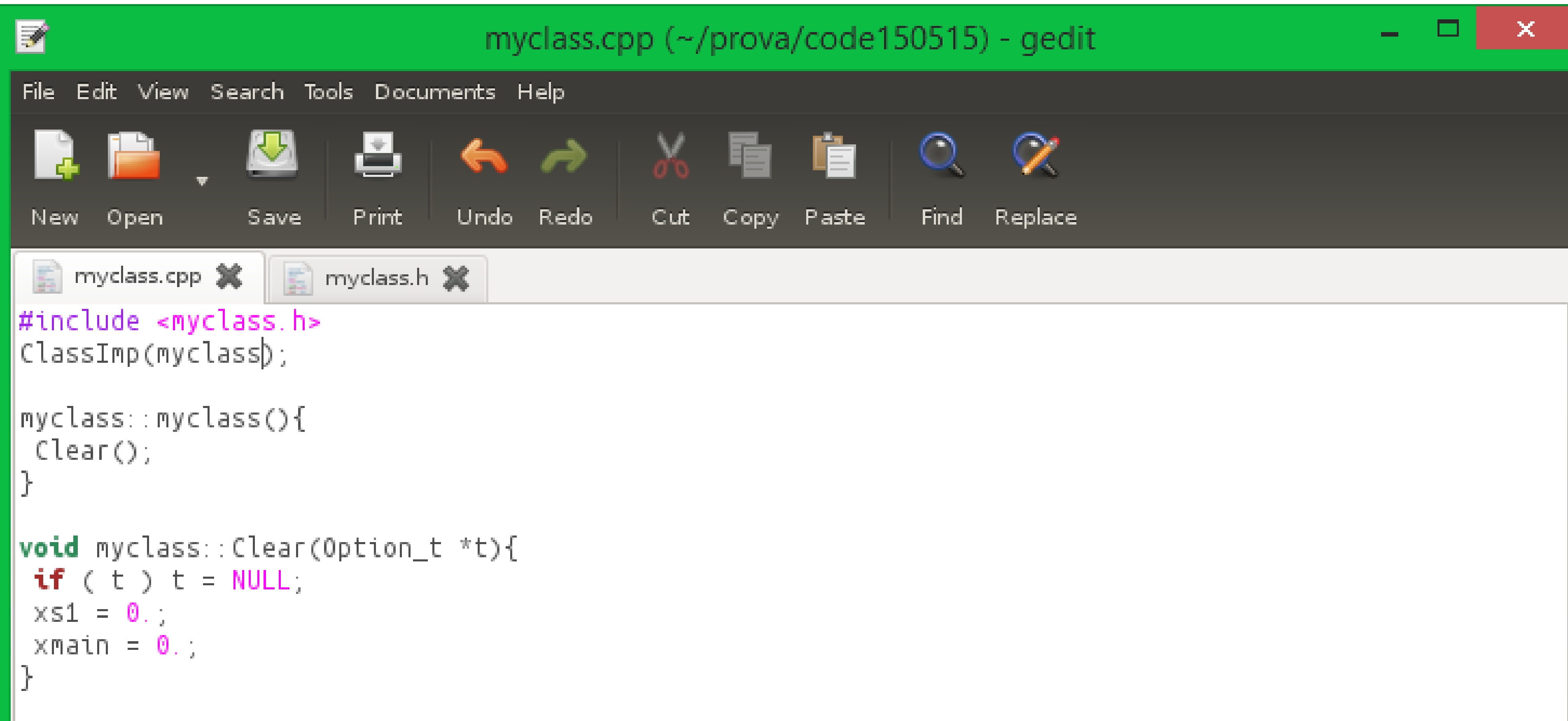
Classes as data containers, example




```
#ifndef myclass_h
#define myclass_h

#include <TObject.h>

class myclass : public TObject{

 public:

  myclass();
  void Clear(Option_t *t=0);

  Float_t xs1;
  Float_t xmain;

  ClassDef(myclass,1);
};

#endif
```

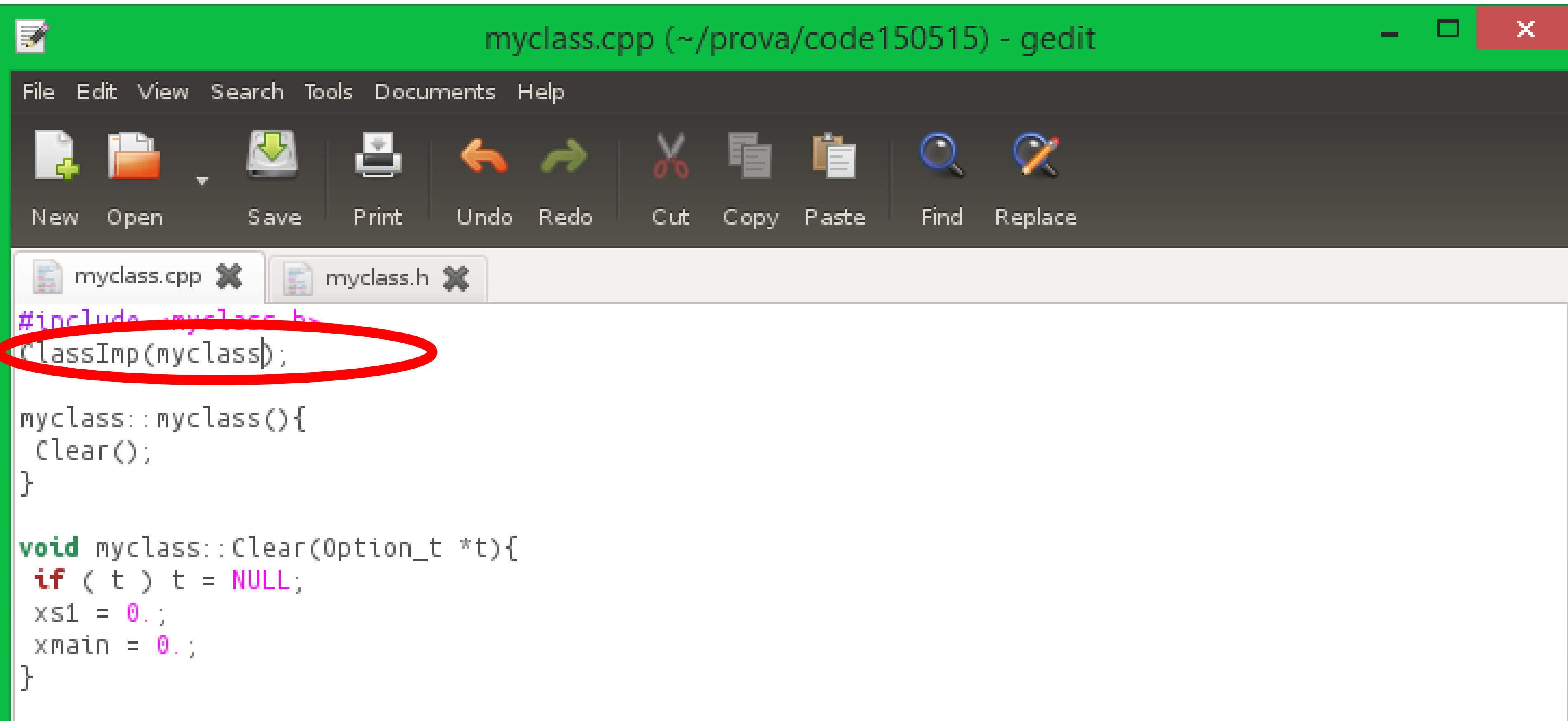
myclass.h (code on Moodle)

Classes as data containers, example



myclass.cpp (code on Moodle)

Classes as data containers, example



myclass.cpp (code on Moodle)

Classes as data containers, how to save a class in a TTree

Saving single variables in a TTree:

```
// create tree
TTree *tree = new TTree("mytree","Example of a TTree");
Double_t xs1 = 0.;
Double_t xmain = 0.;
tree->Branch("xs1",&xs1,"xs1/D");
tree->Branch("xmain",&xmain,"xmain/D");
```

Saving a class in a TTree:

```
TTree *mytree = new TTree("mytree","Example tree");
myclass *pclass = new myclass();
mytree->Branch("mybranch","myclass",&pclass);
```

Classes as data containers, how to store and fill data

single variables:

```
xmain = random->Gaus(-2,1.5);
xs1 = random->Gaus(-0.5,0.5);

tree->Fill();
```

a class:

```
pclass->xmain = random->Gaus(-2,1.5);
pclass->xs1 = random->Gaus(-0.5,0.5);

mytree->Fill();
```

Classes as data containers, how to read the data

single variables:

```
// Open the file
TFile *file = TFile::Open(inputFile);
TTree *tree = (TTree*)file->Get("mytree");

Double_t xs1 = 0.;
Double_t xmain = 0.;
tree->SetBranchAddress("xs1",&xs1);
tree->SetBranchAddress("xmain",&xmain);

for ( Int_t i=0; i<tree->GetEntries(); i++) {

  tree->GetEntry(i);

  main->Fill(xmain);
  s1->Fill(xs1);
```

a class:

```
// Open the file
TFile *file = TFile::Open(inputFile);
TTree *tree = (TTree*)file->Get("mytree");

myclass *pcl = new myclass();

tree->SetBranchAddress("myclass",&pcl);

for (Int_t i=0; i<tree->GetEntries(); i++) {

 tree->GetEntry(i);

 main->Fill(pcl->xmain);
 s1->Fill(pcl->xs1);

}
```

Classes as data containers, how to read the data from CINT

single variables and a class (the same):

```
root[0] TFile *f = TFile::Open(“prova.root”)
root[1] f->ls()
 TFile**          prova.root
 TFile*           prova.root
  KEY: TTree      mytree;1          Example tree
root [2] mytree->Draw(“xs1”)
```

TH2D class

- http://root.cern.ch/root/html532/TH2D.html

In ROOT, the histogram classes are named THdp where d(dimension) can be 1, 2 or 3. The precision p can be C (for Char), S (for Short), F (for Float) or D (for Double).

TH2D is a class for 2-dimensional histograms where a Double_t is used to increment the sum of weights for each cell.

TH2D class, creating a histogram

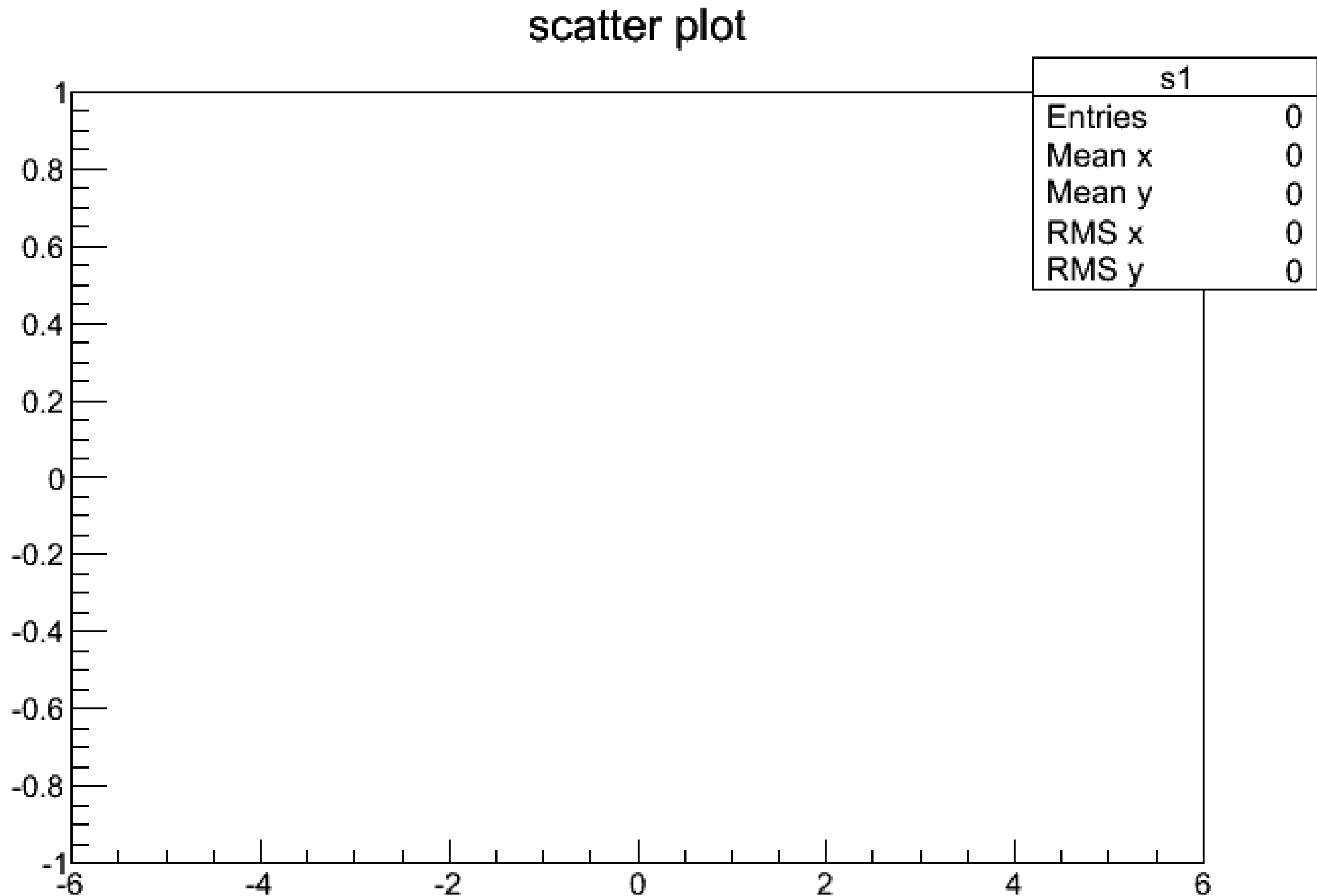
A histogram is created by invoking one of the class constructors. For example:

```
TH2D::TH2D(TString,TString,Int_t,Float_t,Float_t,Int_t,Float_t,
Float_t);

TH2D *h = new TH2D("ROOTname","histogram
title",nbinsx,xlow,xup,nbinsy,ylow,yup);

TH2D *s1 = new TH2D("s1","scatter plot",100,-6.,6.,500,-1.,1.);
```

s1 is a scatter plot with range from -6 to 6 divided into 100 bins on the x and range from -1 to 1 divided into 500 bins on the y.

TH2D class, creating a histogram

```
TH2D *s1 = new TH2D("s1","signal",100,-6.,6.,500,-1.,1.);
```



TH2D class, filling a histogram

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

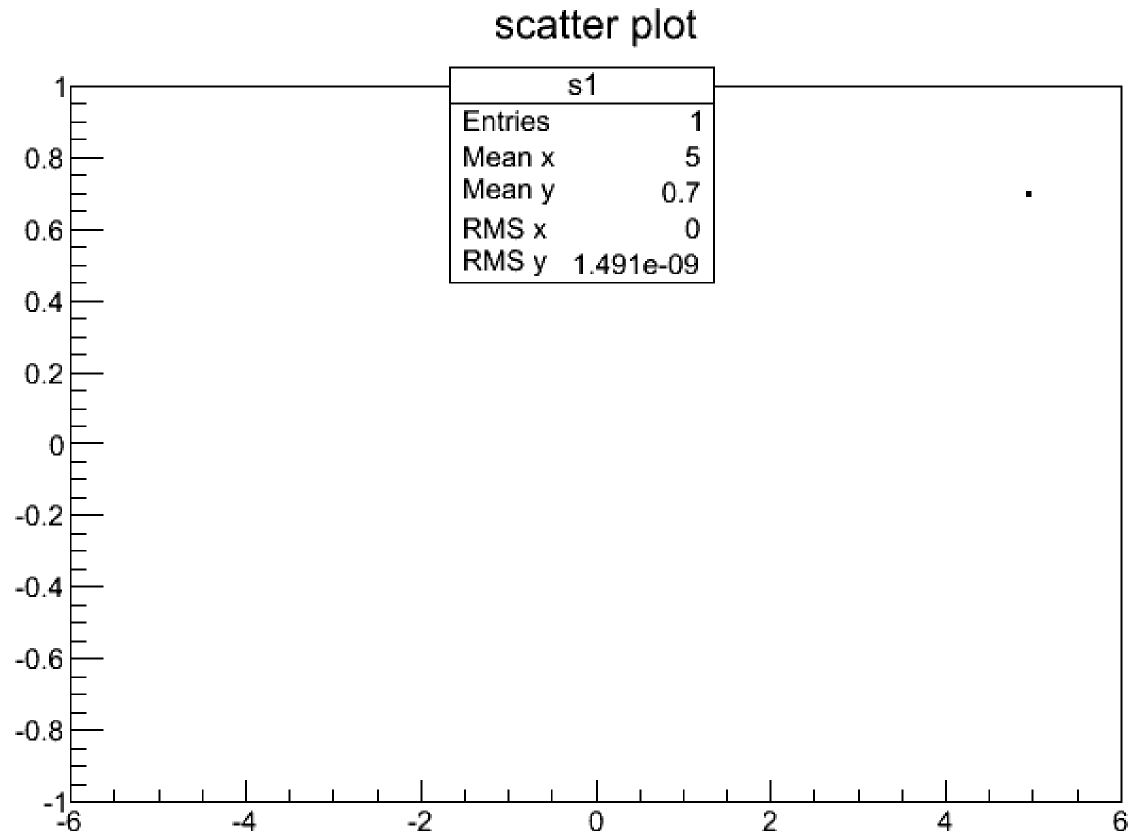
`TH2::Fill(x,y)` increment by `1` the cell corresponding to `x,y` bins.

`TH2::Fill(x,y,w)` increment by `w` the cell corresponding to `x,y` bins.

```
TH2D *s1 = new TH2D("s1","scatter plot",100,-6.,6.,500,-1.,1.);
s1->Fill(5.,0.7);
s1->Fill(-5.,-0.3,666.);
```

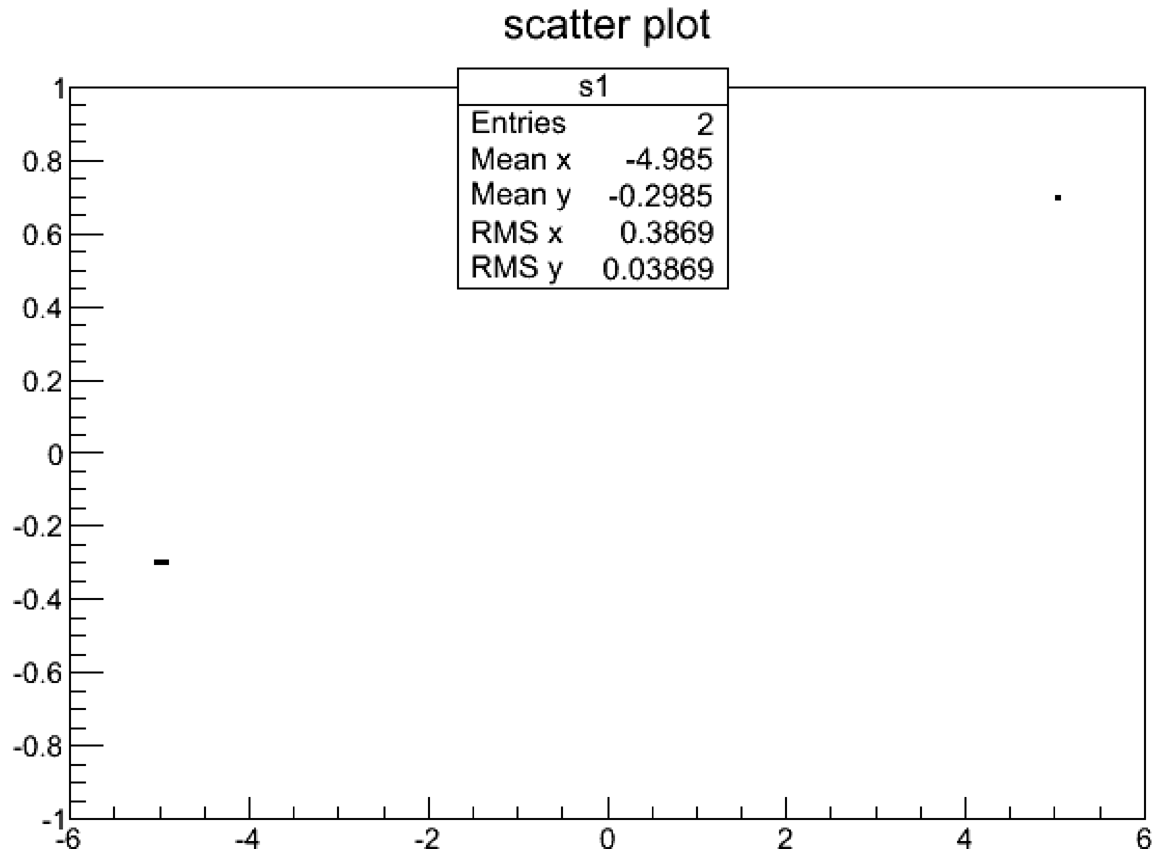
TH2D class, filling a histogram

```
TH2D *s1 = new TH2D("s1","scatter plot",100,-6.,6.,500,-1.,1.);
s1->Fill(5.,0.7);
s1->Draw();
```



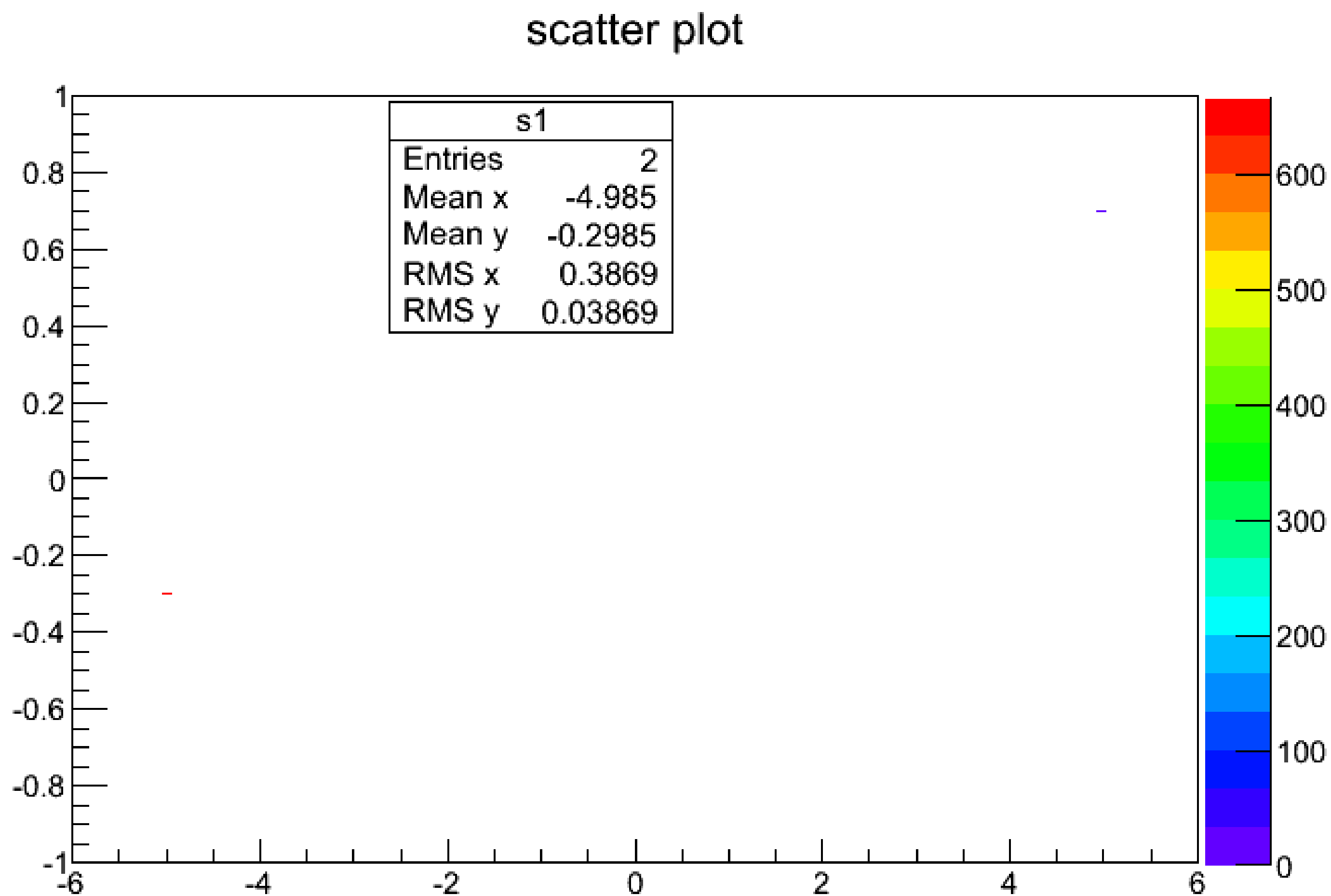
TH2D class, filling a histogram

```
TH2D *s1 = new TH2D("s1","scatter plot",100,-6.,6.,500,-1.,1.);
s1->Fill(5.,0.7);
s1->Fill(-5.,-0.3,666.);
s1->Draw();
```



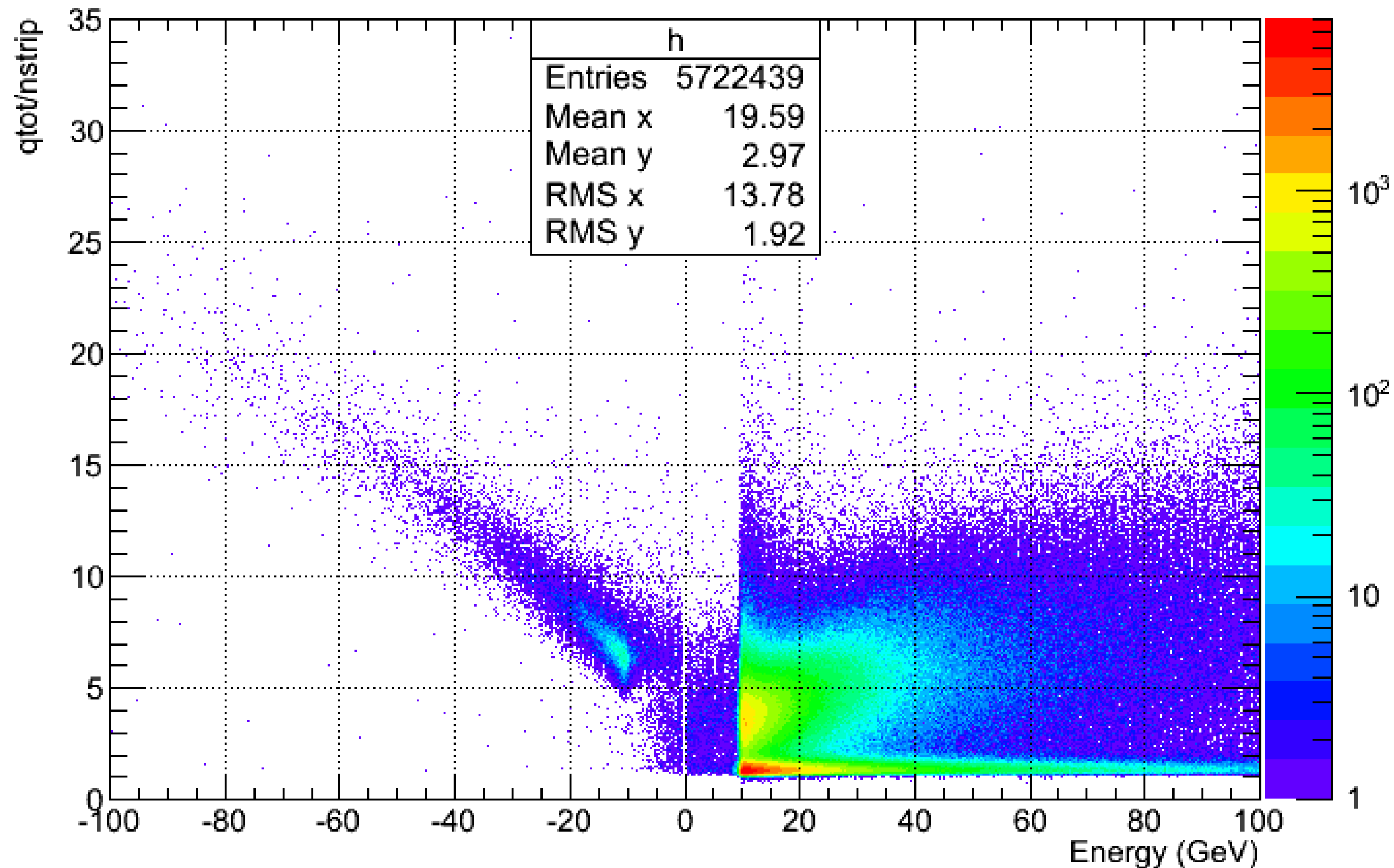
TH2D class, filling a histogram

```
TH2D *s1 = new TH2D("s1","scatter plot",100,-6.,6.,500,-1.,1.);
s1->Fill(5.,0.7);
s1->Fill(-5.,-0.3,666.);
s1->Draw ("colz");
```



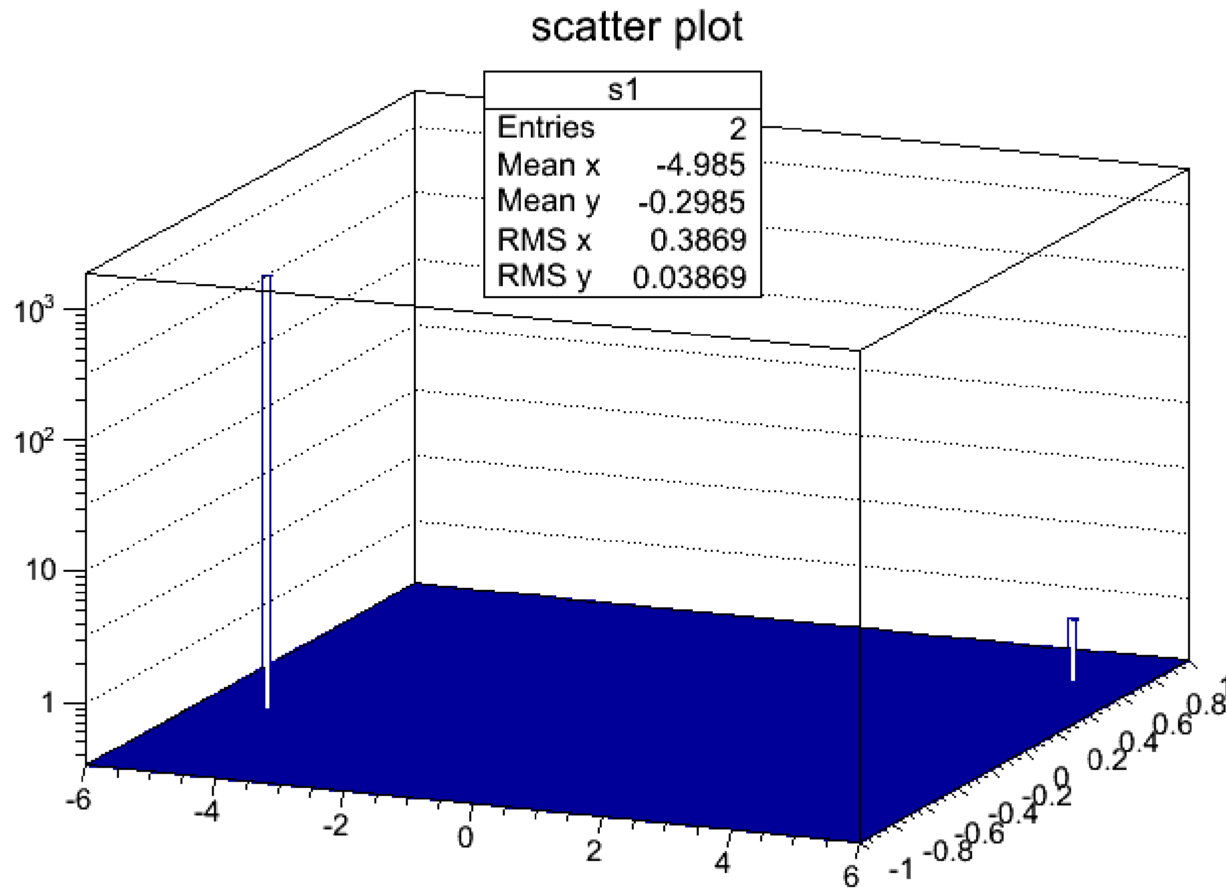
TH2D class, filling a histogram

another plot: “colz” (color + palette)and logZ scale:



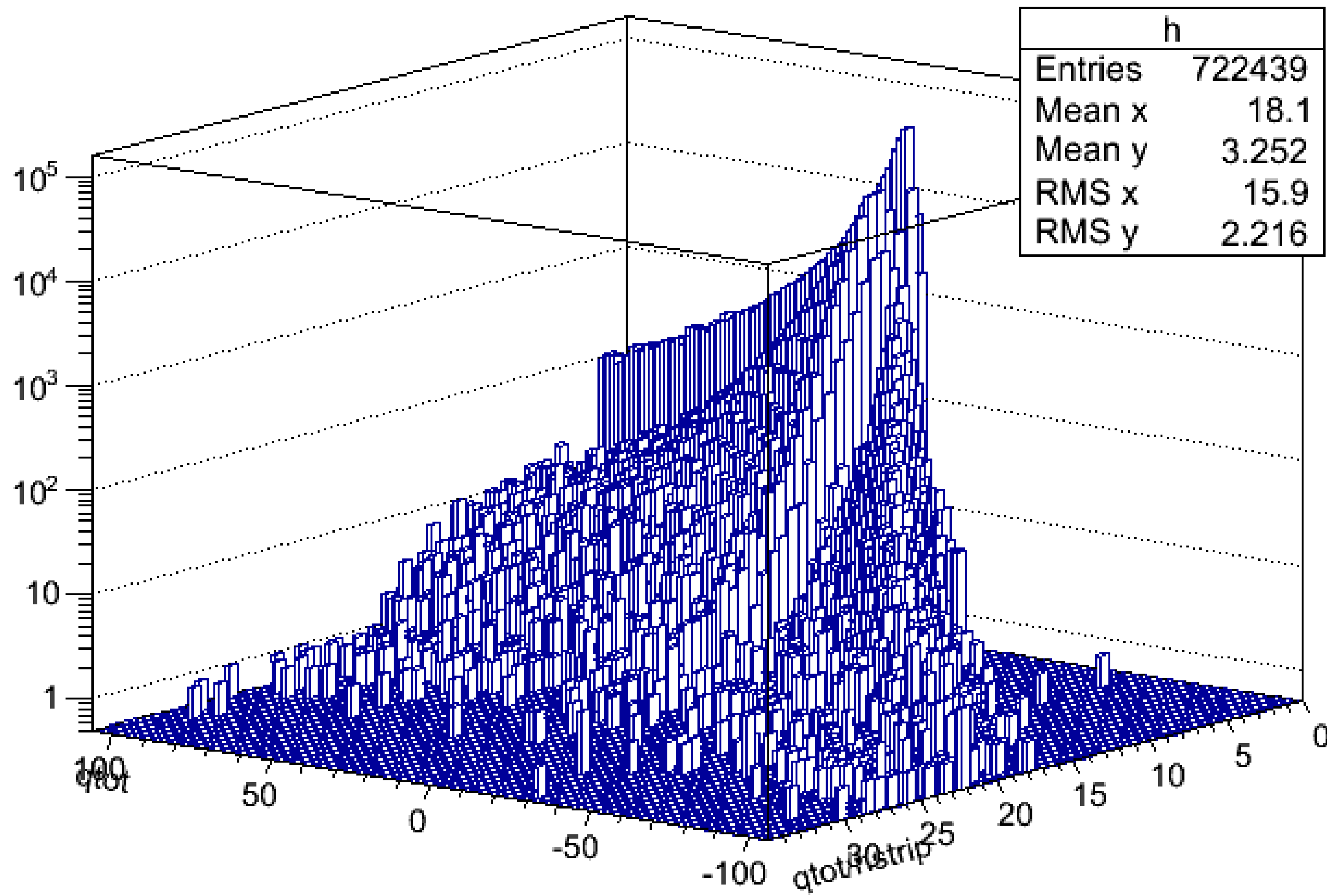
TH2D class, filling a histogram

```
TH2D *s1 = new TH2D("s1","scatter plot",100,-6.,6.,500,-1.,1.);
s1->Fill(5.,0.7);
s1->Fill(-5.,-0.3,666.);
s1->Draw ("lego");
```



TH2D class, filling a histogram

another "lego" example



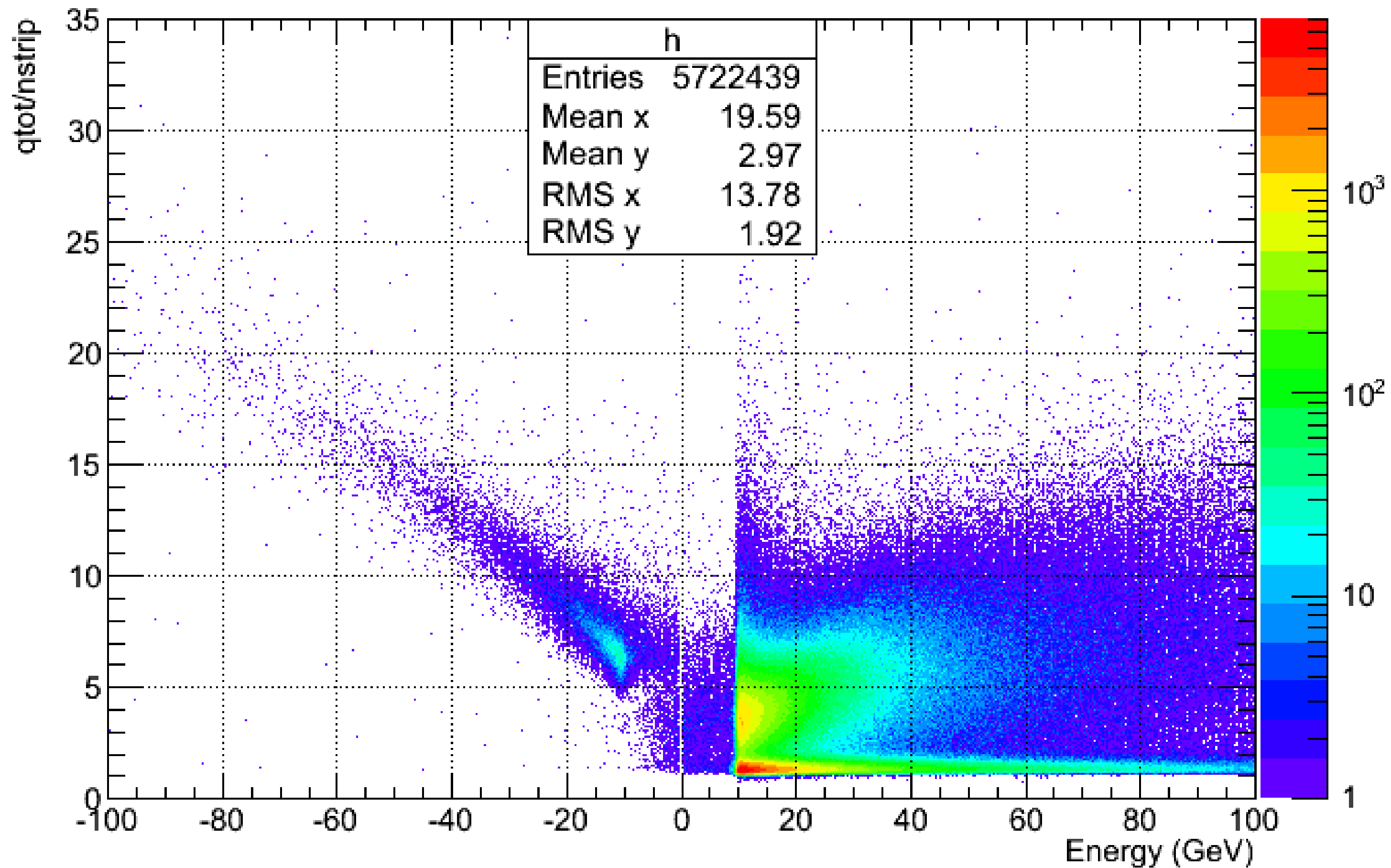
TProfile class

- http://root.cern.ch/root/html532/TProfile.html

Profile histograms are used to display the mean value of Y and its error for each bin in X.

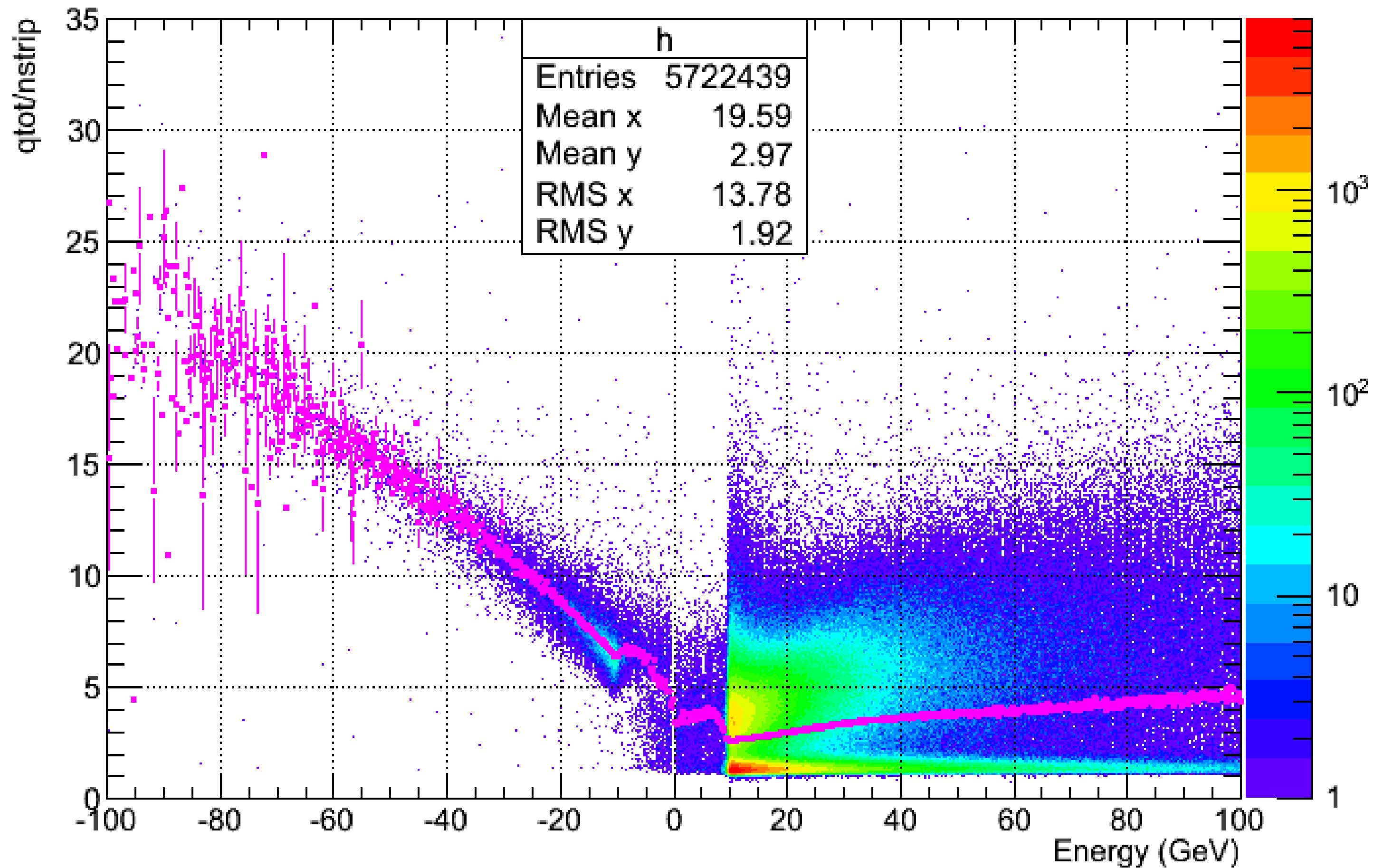
The displayed error is by default the standard error on the mean (i.e. the standard deviation divided by the sqrt(n))

TProfile from a TH2



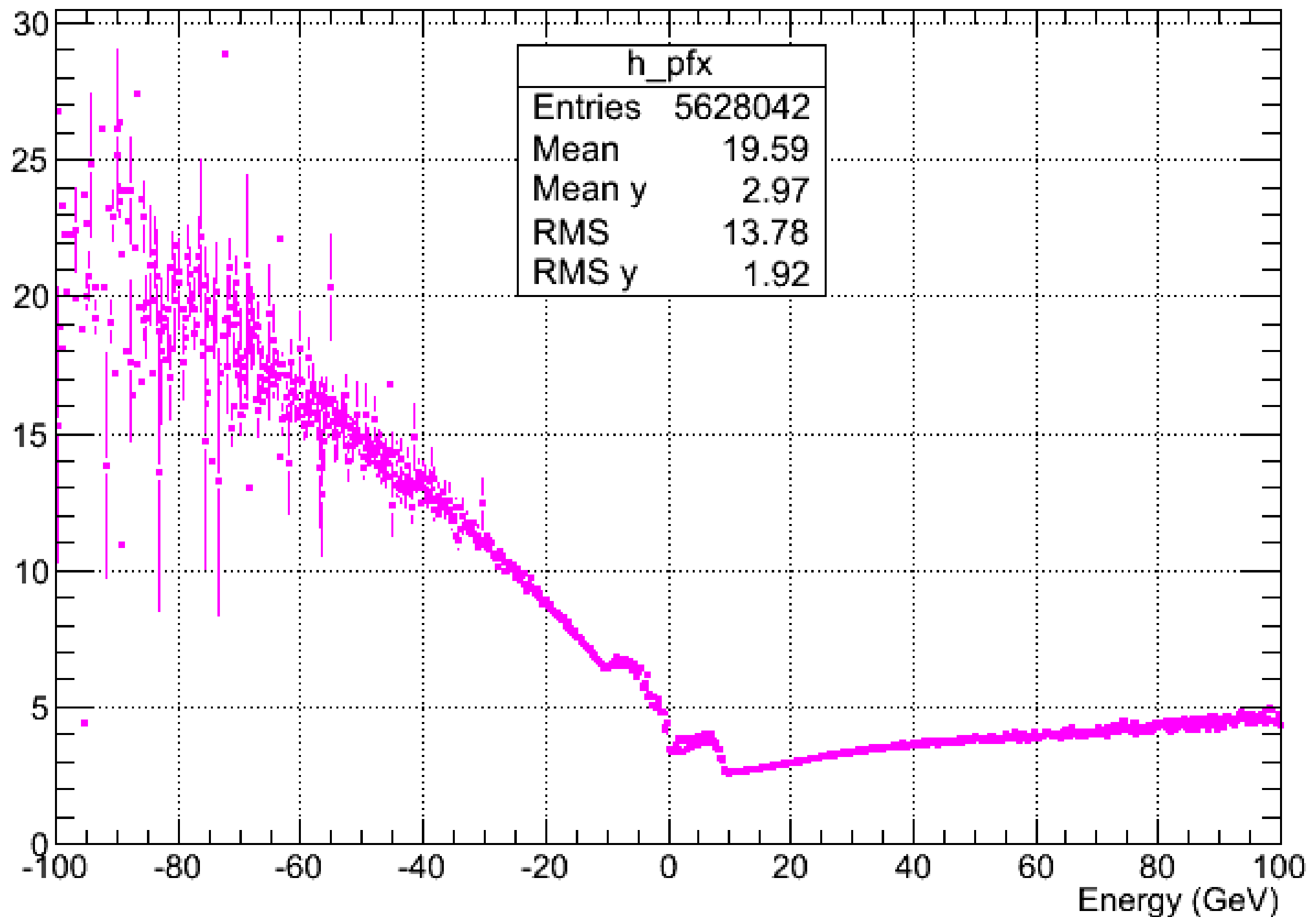
TProfile from a TH2

```
TProfile *prof = h->ProfileX();
prof->Draw("same");
```



TProfile from a TH2

```
TProfile *prof = h->ProfileX();
prof->Draw();
```



TGraph class

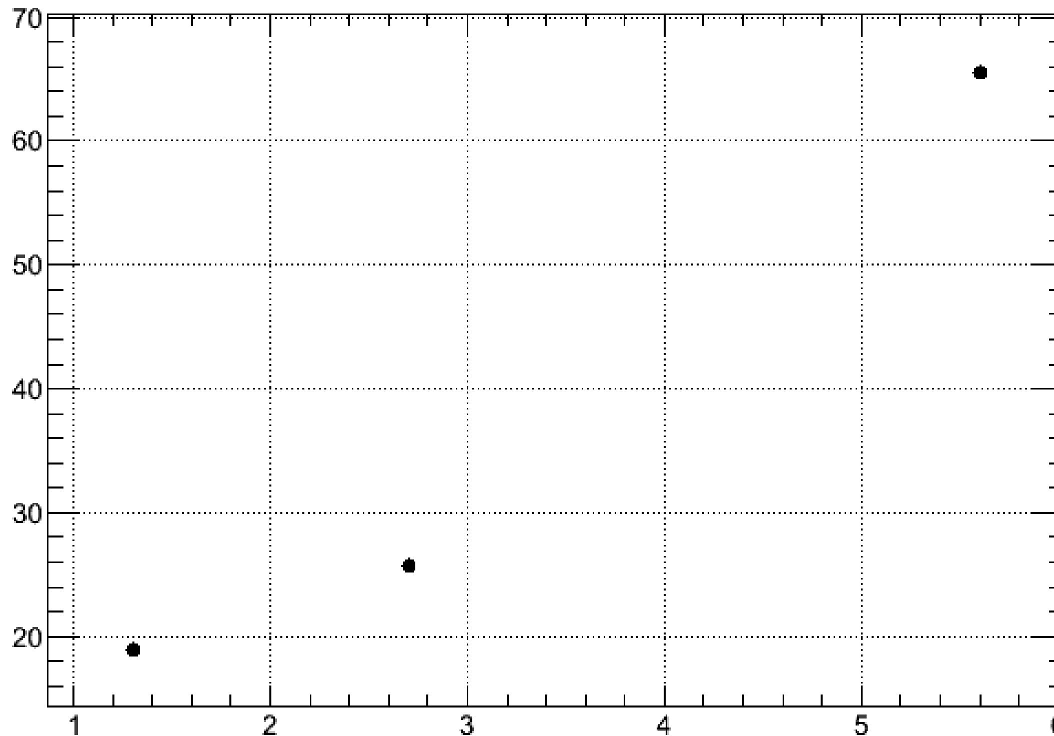
- http://root.cern.ch/root/html532/TGraph.html

A TGraph is a graphics object made of two arrays X and Y with npoints each.

We use TGraph (and/or TGraphErrors when we need also the X and Y associated errors) to plot data in a certain position given by the pairs (Xi,Yi).

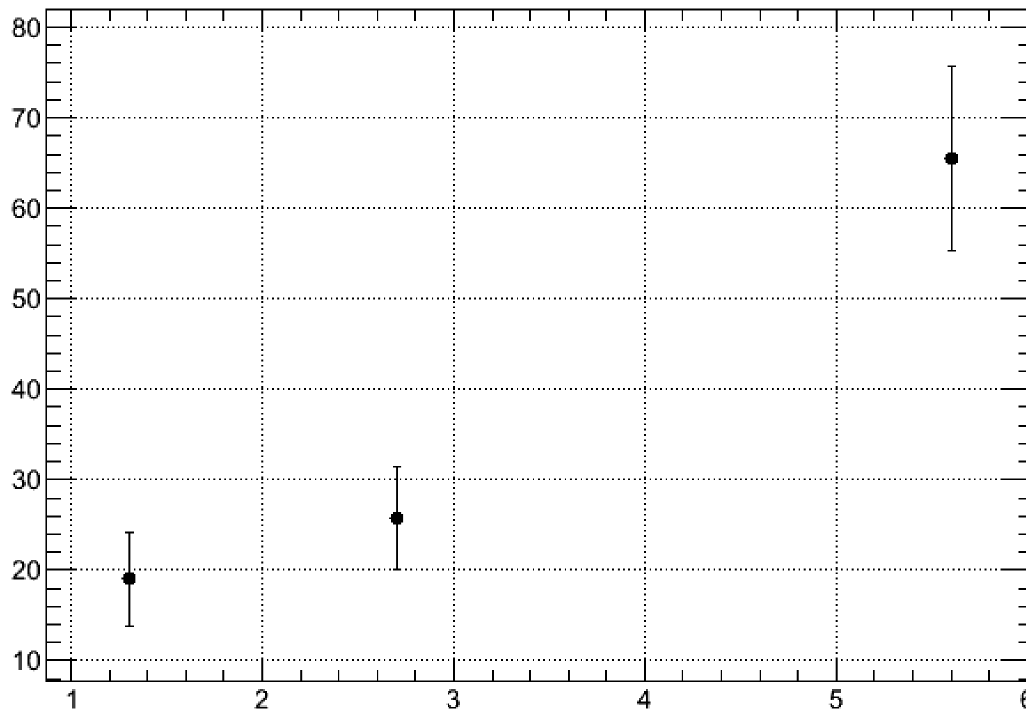
TGraph example

```
Float_t x[3] = {1.3, 2.7,  5.6};
Float_t y[3] = {19., 25.7, 65.6};
TGraph *mygraph = new TGraph(3,x,y);
mygraph->SetMarkerStyle(20);
mygraph->Draw ("AP"); // A = draw axis, P = draw points
```



TGraphErrors example

```
Float_t x[3] =  {1.3, 2.7,  5.6};
Float_t y[3] =  {19., 25.7, 65.6};
Float_t ey[3] = {5.1, 5.7,  10.2};
TGraphErrors *mygraph = new TGraphErrors(3,x,y,NULL,ey);
mygraph->SetMarkerStyle(20);
mygraph->Draw ("AP"); // A = draw axis, P = draw points
```



Collection classes

A collection is a group of related objects.

You will find it easier to manage a large number of items as a collection. For example, a diagram editor might manage a collection of points and lines. A set of widgets for a graphical user interface can be placed in a collection. A geometrical model can be described by collections of shapes, materials and rotation matrices. Collections can themselves be placed in collections. Collections act as flexible alternatives to traditional data structures of computers science such as arrays.

https://root.cern.ch/root/htmldoc/guides/users-guide/CollectionClasses.html

Collection classes properties

The ROOT collections are *polymorphic* containers that hold pointers to TObjects, so:

- They can **only** hold objects that inherit from TObject
- They return pointers to TObjects, that have to be cast back to the correct subclass

Collections are dynamic; they can grow in size as required. Collections themselves are descendants of TObject so can themselves be held in collections. It is possible to nest one type of collection inside another to any level to produce structures of arbitrary complexity.

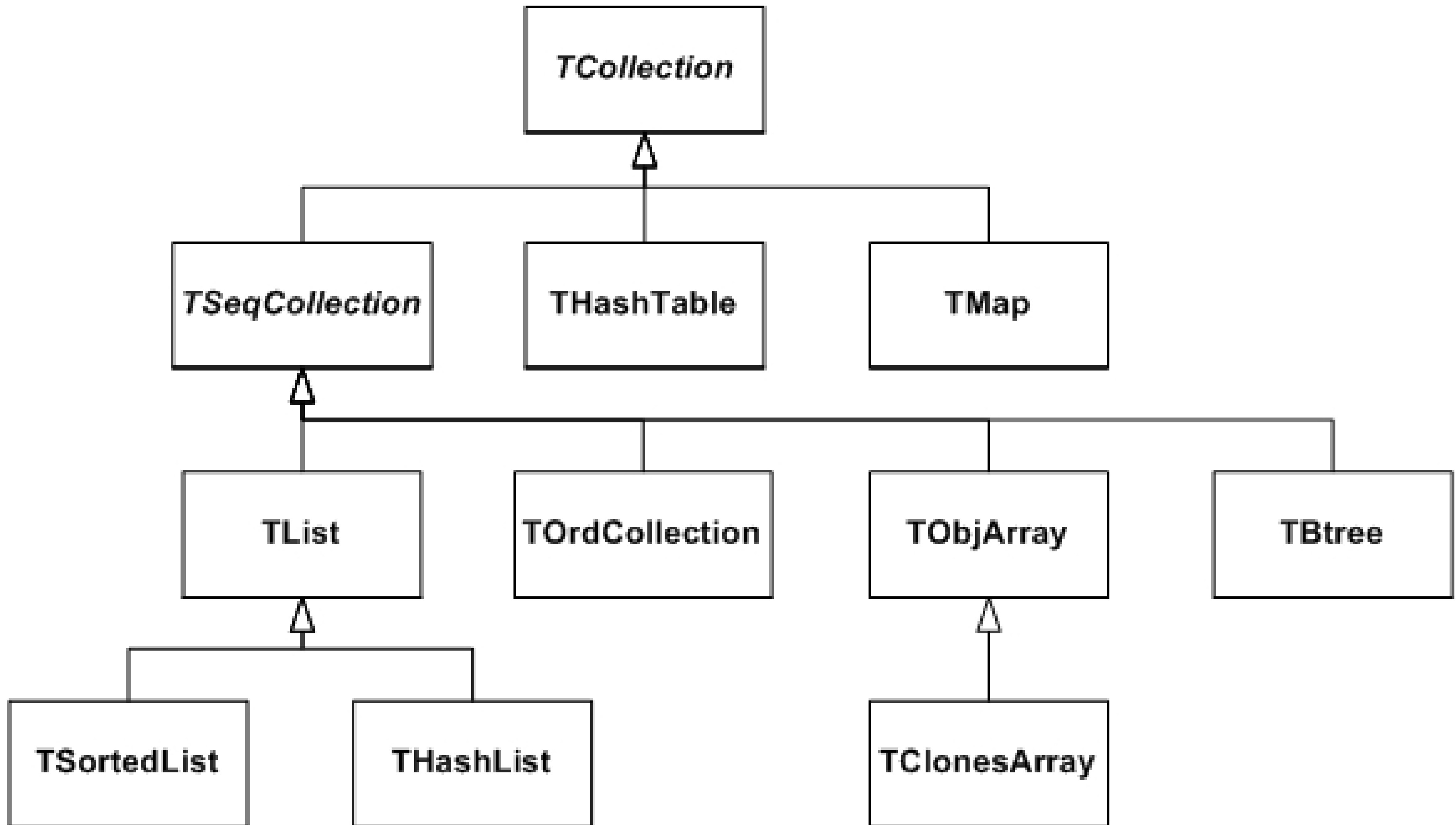
Collection classes ownership

The ROOT collections **do not own** the objects they hold for the very good reason that the same object could be a member of more than one collection.

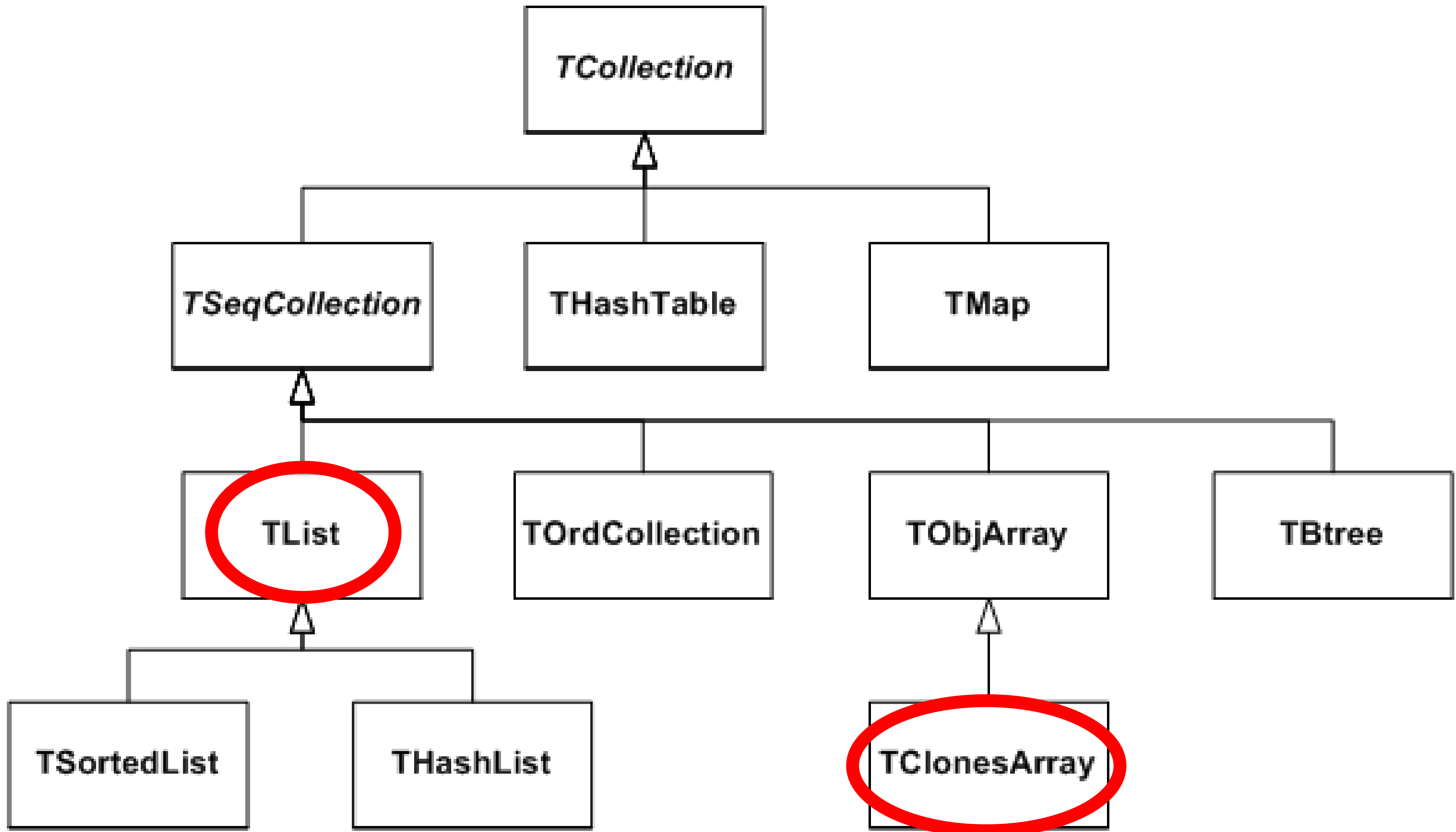
Object ownership is important when it comes to deleting objects; if nobody owns the object it could end up as wasted memory (i.e. a memory leak) when no longer needed.

If a collection is deleted, its objects are not. The user can force a collection to delete its objects, but that is the user's choice.

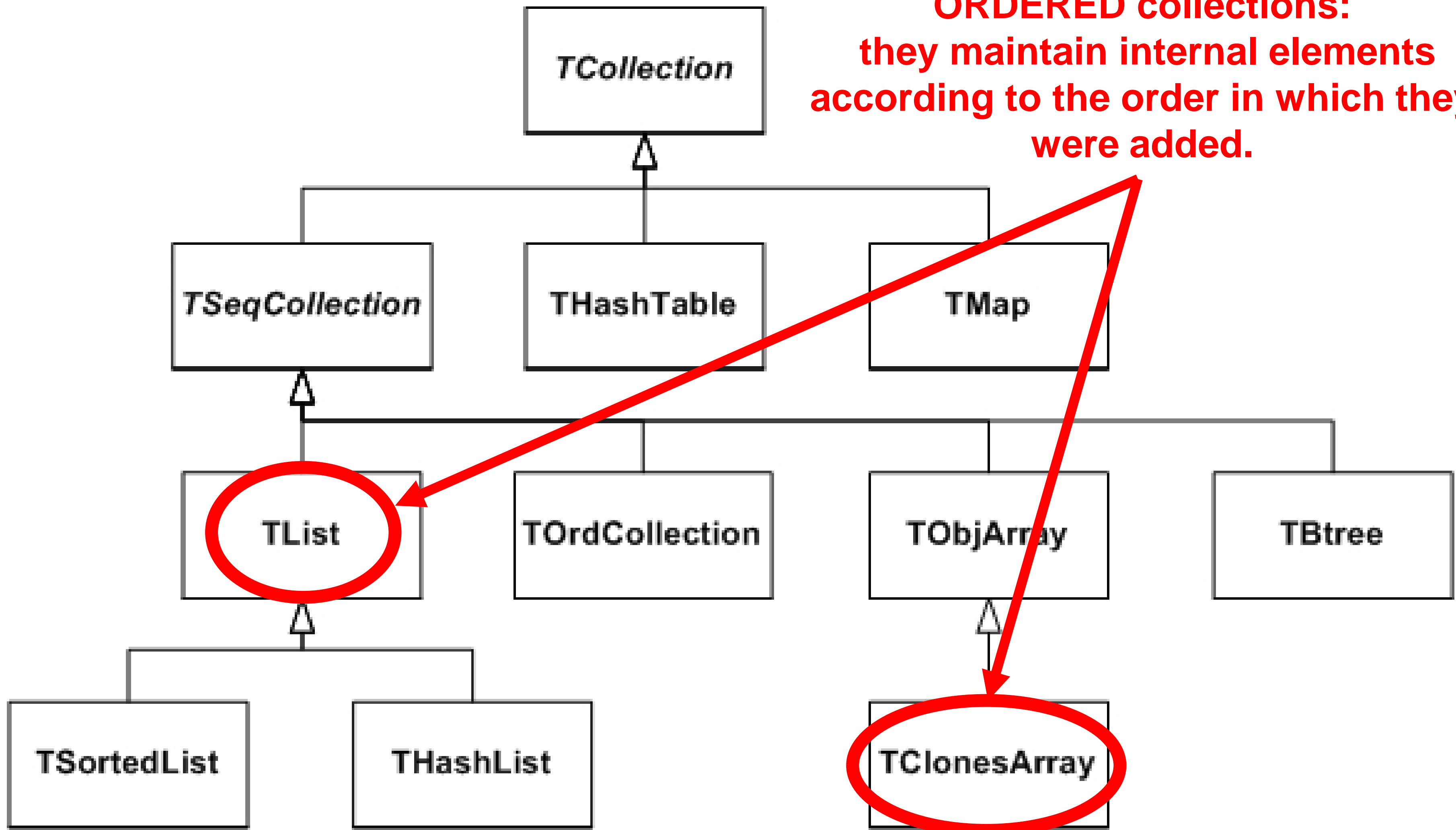
Types of collections



Types of collections



Types of collections



ORDERED collections: they maintain internal elements according to the order in which they were added.

Collectable objects

By default, **all objects of TObject derived classes can be stored in ROOT containers**.

The TObject class provides some member functions that allow you to tune the behavior of objects in containers. For example, by default two objects are considered equal if their pointers point to the same address. This might be too strict for some classes where equality is already achieved if some or all of the data members are equal. By overriding some (IsEqual(), Compare(), IsSortable, Hash()) TObject member functions, you can change the behavior of objects in collections:

TList

A TList is a doubly linked list*. Before being inserted into the list the object pointer is wrapped in a TObjLink object that contains, besides the object pointer also a previous and next pointer.

Objects are typically added using:

`Add()`

`AddFirst(), AddLast()`

`AddBefore(), AddAfter()`

Main features of TList: very low cost of adding/removing elements anywhere in the list.

* a doubly linked list is a linked data structure that consists of a set of sequentially linked records called nodes. Each node contains two fields, called links, that are references to the previous and to the next node in the sequence of nodes.

TList example

```
TH2D *h2 = new TH2D("h2","h2",100,0.,1.,100,0.,1.);
TH1D *h1 = new TH1D("h1","h1",100,0.,1.);

TList *mylist = new TList();
mylist->Add(h2);
mylist->Add(h1);

mylist->At(0)->Draw(); // draw h2


// open file

mylist->Write("list",1);

// close file
```

Iteration

There are several ways to iterate over a TList:

Using the TList iterator TListIter (via the wrapper class TIter):

```
TIter next(mylist);
while ( TObject *obj = next() ){
   obj->Draw();
}
```

Using the TObjLink list entries (that wrap the TObject*):

```
TObjLink *lnk = GetListOfPrimitives()->FirstLink();
while (lnk) {
   lnk->GetObject()->Draw();
   lnk = lnk->Next();
}
```

with a loop

```
for (Int_t i=0; i < mylist->GetEntries(); i++ ){
  mylist->At(i)->Draw();
}
```

TList example

```
TH2D *h2 = new TH2D("h2","h2",100,0.,1.,100,0.,1.);
TH1D *h1 = new TH1D("h1","h1",100,0.,1.);

TList *mylist = new TList();
mylist->Add(h2);
mylist->Add(h1);

mylist->At(0)->Draw(); // draw h2

TCanvas *c[2];
for (Int_t i=0; i < mylist->GetEntries(); i++ ){
  c[i] = new TCanvas();
  mylist->At(i)->Draw();
}
```

TClonesArray

A TClonesArray is an array of identical (clone) objects. The memory for the objects stored in the array is allocated only once in the lifetime of the clones array. All objects must be of the same class.

TClonesArray supports traditional array semantics via the overloading of operator[ ]. Objects can be directly accessed via an index. The array expands automatically when objects are added. At creation time one specifies the default array size (default = 16) and lower bound (default = 0). Resizing involves a re-allocation and a copy of the old array to the new. This can be costly if done too often. If possible, set initial size close to expected final size. Index validity is always checked (if you are 100% sure and maximum performance is needed you can use UnCheckedAt() instead of At() or operator[ ]). If the stored objects are sort able the array can be sorted using Sort().

TClonesArray example

```
TClonesArray *arr = new TClonesArray("TTrack");

for (Int_t ev=0;ev<10000;ev++) { // loop over the events
 ...
 arr->Clear();
 for (Int_t i=0;i<numberoftracks;i++){// loop over tracks
        TTrack *track = new TTrack();
        ...
        track->abc = abc;
        ...

        new((*arr)[i]) TTrack(*track);
 }
```

TClonesArray example

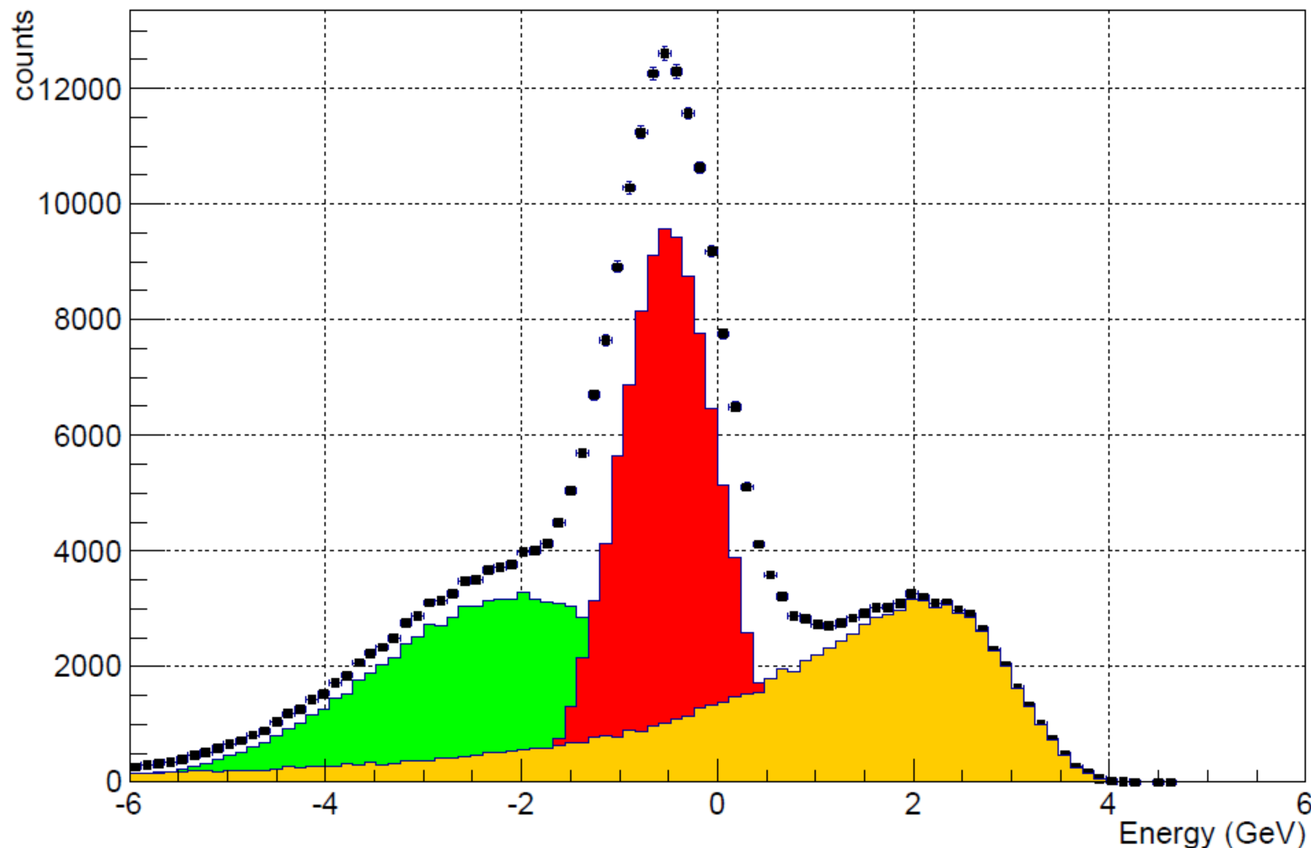
```
TClonesArray a("TTrack", 10000);

//read a

while (TEvent *ev = (TEvent *)next()) {       // O(100000)
   for (int i = 0; i < ev->Ntracks; i++) {    // O(10000)
      TTrack *track = (TTrack*)a.ConstructedAt(i);
      track->Set(x,y,z,...);
      ...
   }
   ...
   a.Clear();
}
```

Exercises

6. once you fixed the bug in `treeExample2.C` , change the script in order to:
   - remove useless variable(s);
   - add two new random generated gaussian variables (red and green distribution of the previous lesson exercise).

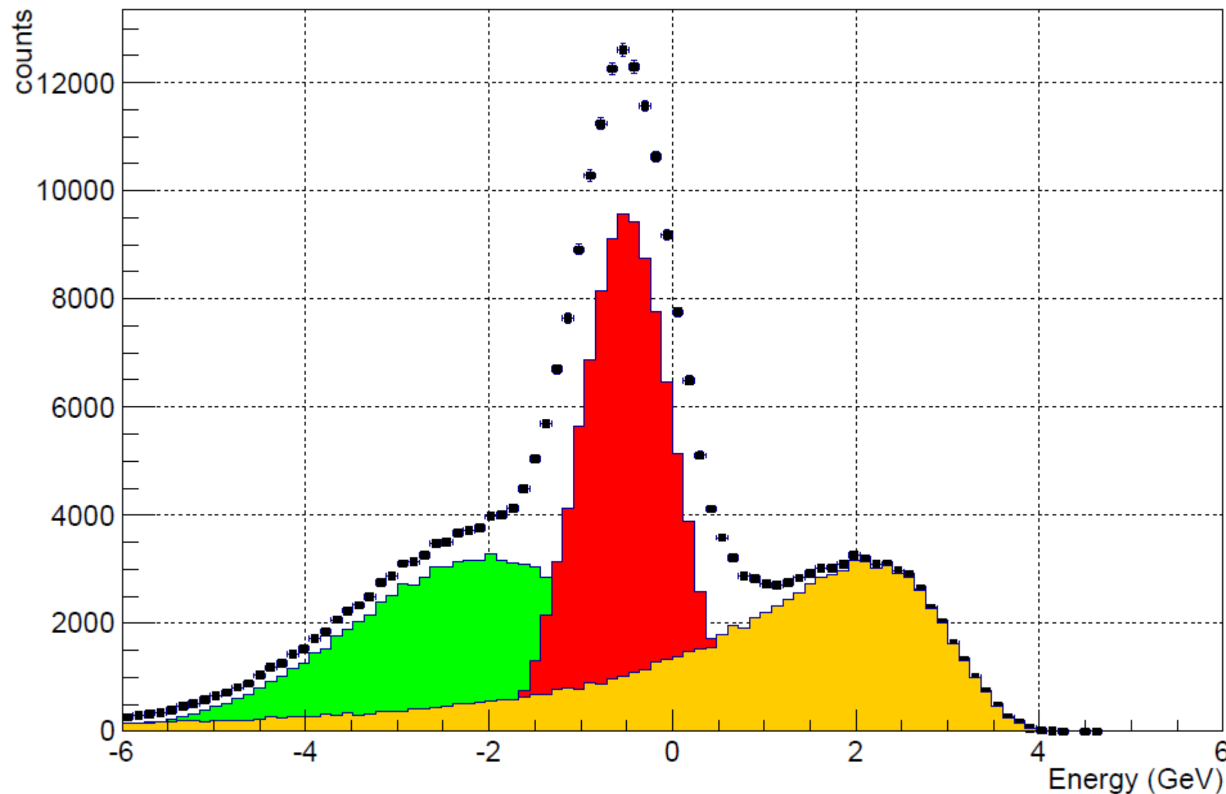
Exercises

7. write a `readTree2.C` script in order to re-create the last time histogram by reading data from the `treeExample2.C` output file:



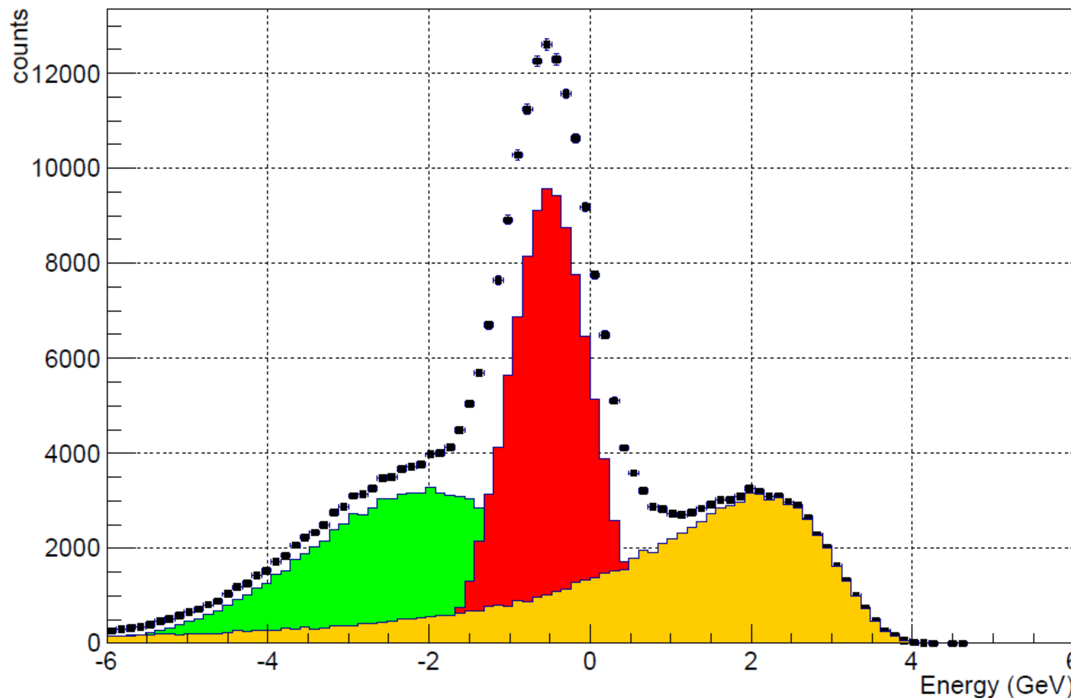
Exercises

8. same as points 6. and 7. but using a class (myclass) instead of single variables (hints: you must first compile the class from CINT, .L myclass.cpp++)



Exercises

9. for each bin of the green histogram print on the STDOUT a row with two columns containing the value of the center of the bin and the content of the bin (the number of counts), hints: look at the documentation of the class TH1



Exercises

10. Create a 2D histogram (scatter plot) of the red vs green distribution of the previous plot.
11. Get the ProfileX from the 2D histogram.
12. Save histograms, canvas and profile into a file using a single ordered list (TList).