Class Descriptions

The full reference guide of all ROOT classes is available on the web via the URL:

- `http://root.cern.ch/root/html/ClassIndex.html`

Also reachable via the item ‘‘**Classes and Members Reference Guide**’’ on the ROOT home page.

The class header files are always the same as the class name + .h:

- `TROOT          #include <TROOT.h>`
- `TObject        #include <TObject.h>`
- `TServerSocket  #include <TServerSocket.h>`

The **TROOT** Class - The ROOT of Everything

The **TROOT** object is the entry point to the system.

The single instance of `TROOT` is always accessible via the global `gROOT`. Using the `gROOT` pointer one has access to basically every object created in a ROOT based program.

The following lists are accessible via **gROOT**:

- `gROOT->GetListOf`**`Classes`**`()`
- `gROOT->GetListOf`**`Types`**`()`
- `gROOT->GetListOf`**`Globals`**`()`
- `gROOT->GetListOf`**`GlobalFunctions`**`()`
- `gROOT->GetListOf`**`Files`**`()`
- `gROOT->GetListOf`**`Sockets`**`()`
- `gROOT->GetListOf`**`Canvases`**`()`
- `gROOT->GetListOf`**`Styles`**`()`
- `gROOT->GetListOf`**`Colors`**`()`
- `gROOT->GetListOf`**`Functions`**`()`
- `gROOT->GetListOf`**`Geometries`**`()`
- `gROOT->GetListOf`**`Browsers`**`()`

The **TROOT** class also provides many useful services, like:

- Get (`GetFile()`) or find (`FindObject()`) a pointer to an object in any of the above lists
- General utilities: `Time()`, `Idle()`, `Macro()`, `ProcessLine()`, `Reset()`, etc.

The `TObject` Class - The Mother of All Objects

The `TObject` class provides default behaviour and protocol for all objects in the ROOT system.

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

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

A `TObject` has two 4 byte data members:

- `UInt_t fBits` (bit field status word)
- `UInt_t fUniqueID` (object unique identifier)

Of `fBits` the high 8 bits are reserved by the system and the low 24 bits are user setable. `fUniqueID` is unused for the time being

Every object which inherits from TObject can be stored in the ROOT collection classes.

Intermezzo: Machine Independent Basic Types

The ROOT system defines a set of typedefs for the basic types, like int, short, long, etc. Use these typedefs to be sure your code is, and will stay, portable:

• `Char_t`	Signed 1 byte
• `UChar_t`	Unsigned 1 byte
• `Short_t`	Signed short integer 2 bytes
• `UShort_t`	Unsigned short integer 2 bytes
• `Int_t`	Signed integer 4 bytes
• `UInt_t`	Unsigned integer 4 bytes
• `Long_t`	Signed long integer 8 bytes
• `ULong_t`	Unsigned long integer 8 bytes
• `Float_t`	Float 4 bytes
• `Double_t`	Float 8 bytes
• `Bool_t`	Boolean (`kFALSE`, `kTRUE`)

The ROOT Collection Classes

Collections are a key feature of the ROOT system. Many, if not most, of the applications you write will use collections. The ROOT collections are so called *polymorphic collections*.

The following features will be demonstrated:

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

Understanding Collections

A collection is a group of related objects:

- Collections of points and lines might be managed by a graphics pad
- A vertex will have a collection of tracks
- A detector geometry contains collections of shapes, materials, rotation matrices and sub-detectors

Collections can be thought of as polymorphic containers that can contain different types of elements:

- Elements must be instances of classes
- Elements must be instances of classes descending from **`TObject`**

Collections themselves are descendants of `TObject`. So collections can contain other collections in a tree structure:

- Graphics pads (`TPad`) containing other pads
- Detectors in detectors

The basic protocol `TObject` defines for collection elements are:

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

How to use and override these members functions will be shown later.

Types of Collections

The ROOT system implements the following type of collections:

Ordered Collections (Sequences)

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

- `TList`
- `THashList`
- `TOrdCollection`
- `TObjArray`
- `TClonesArray`

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

Sorted Collection

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

- `TSortedList`
- `TBtree`

Types of Collections (contd.)

Unordered Collections

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

- `THashTable`
- `TMap`

Iterators

Each collection class has its own associated iterator class.

An iterator object is used to traverse (walk through) a collection. For example:

- `TList`   `TListIter`
- `TMap`   `TMapIter`

In general we will use the **`TIter`** wrapper class.

The `TCollection` Abstract Base Class

The `TCollection` class provides the basic protocol all collection classes have to implement. Like:

- `Add()`, `AddAll()`, `Remove()`, `RemoveAll()`
- `FindObject()`
- `MakeIterator()`
- `Clear()`, `Delete()`

The ROOT collection classes always store pointers to objects that inherit from `TObject`. They never adopt the objects. Therefore, it is the user's responsability to take care of deleting the actual objects once they are not needed anymore. In exceptional cases, when the user is 100% sure nothing else is referencing the objects in the collection, one can delete all objects and the collection at the same time using the **`Delete()`** function. To clear or reset a collection use **`Clear()`** (this clears the collection memory structures but does not delete the objects). `Clear()` typically is also called via the collection destructor.

Typically there is only one owning collection. For example:

```
class TEvent : public TObject {
private:
   TList *fTracks;  //list of all tracks
   TList *fVertex1; //subset of tracks part of vertex1
   TList *fVertex2; //subset of tracks part of vertex2
   ...
};

TEvent::~TEvent()
{
   fTracks->Delete(); delete fTracks;
   delete fVertex1; delete fVertex2;
}
```

A Collectable Class

```
// TObjNum is a simple container for an integer.
class TObjNum : public TObject {
private:
   int  num;

public:
   TObjNum(int i = 0) : num(i) { }
   ~TObjNum() { Printf("~TObjNum = %d", num); }
   void     SetNum(int i) { num = i; }
   int      GetNum() { return num; }
   void     Print(Option_t *)
              { Printf("num = %d", num); }

   Bool_t   IsEqual(TObject *obj)
              { return num == ((TObjNum*)obj)->num; }
   Bool_t   IsSortable() const { return kTRUE; }
   Int_t    Compare(TObject *obj)
              { if (num < ((TObjNum*)obj)->num)
                     return -1;
                else if (num > ((TObjNum*)obj)->num)
                     return 1;
                else
                     return 0; }
   ULong_t  Hash() { return num; }
};
```

`IsEqual()` is used by the `FindObject()` collection method. By default `TObject::IsEqual()` compares the two object pointers

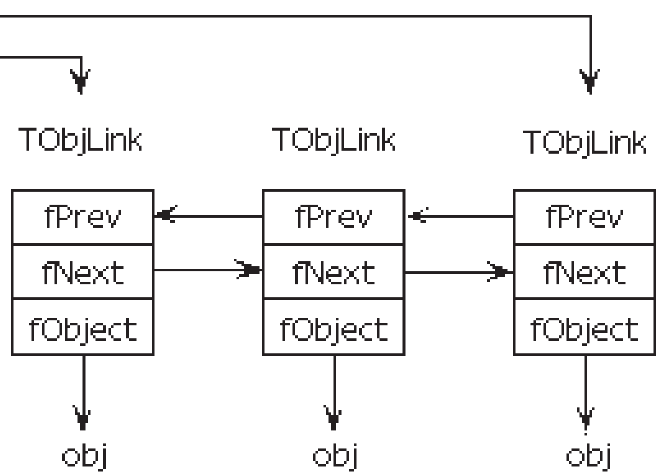
`IsSortable()` and **`Compare()`** need to be implemented by all sortable classes. By default a `TObject` is not sortable.

Hash() needs to be implemented by all objects that need to be stored in a hashtable (`THashTable`, `THashList` and `TMap`). By default `TObject::Hash()` returns the address of the object. It is essential to choose a good hash function.

The **`TList`** Collection

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

```
class TList : public TSeqCollection {
private:
   TObjLink *fLast;
   TObjLink *fFirst;
   •••
   •••
};

class TObjLink {
friend class TList;
private:
   TObjLink *fPrev;
   TObjLink *fNext;
   TObject  *fObject;
   •••
   •••
};
```



Objects are typically added using:

- Add()
- AddFirst(), AddLast()
- AddBefore(), AddAfter()

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

Overhead per element: 1 `TObjLink`, i.e. two (4 byte) pointers + pointer to vtable = 12 bytes.

Iterating over a `TList`

There are basically four ways to iterate over a `TList`:

1 Using the **`ForEach`** macro:

```
GetListOfPrimitives()->ForEach(TObject,Draw)();
```

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

```
TIter next(GetListOfTracks());
while (TTrack *obj = (TTrack *)next())
   obj->Draw();
```

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

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

4 Using the TList's **After()** and **Before()** member functions:

```
TFree *idcur = this;
while (idcur) {
   ...
   ...
   idcur = (TFree*)GetListOfFree()->After(idcur);
}
```

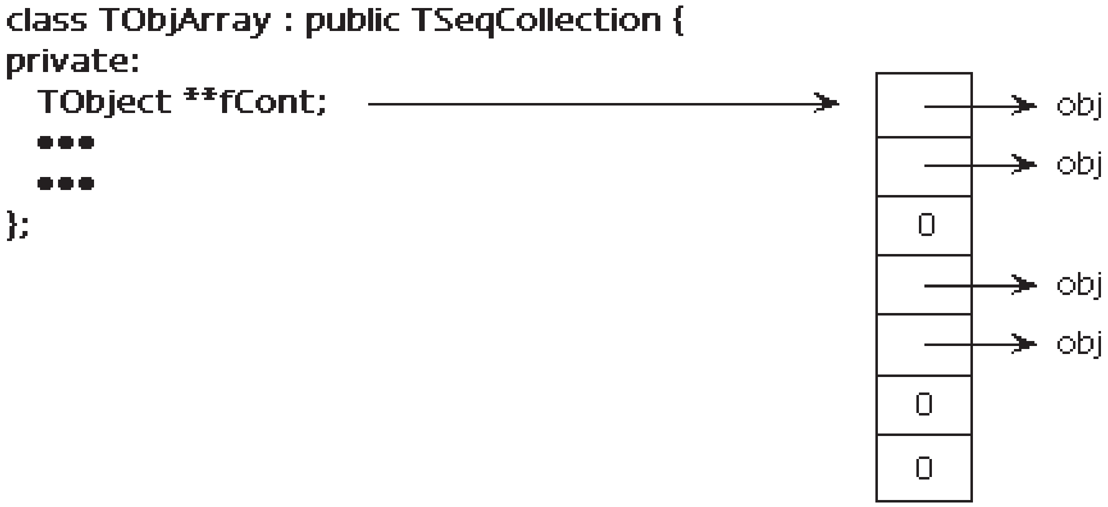
Method 1 uses internally method 2.

Method 2 works for all collection classes. `TIter` overloads **`operator()`**.

Methods 3 and 4 are specific for `TList`.

The **TObjArray** Collection

An array of `TObjects`. The array expands automatically when objects are added. Overloads `operator[]` to get builtin like array semantics.



At creation time specify default array size (default = 16) and lowerbound (default = 0). Resizing involves a re-alloc and a copy of the old array to the new. Can be costly if done too often. If possible, set initial size close to expected final size. Bounds are always checked (if 100% sure and maximum performance needed you can use `UnCheckedAt()` instead of `At()` or `operator[]`).

If stored objects are sortable array can be sorted (using `Sort()`). Once sorted, efficient searching using the `BinarySearch()` method.

Iterate using:

1 **`TIter`**

2
```
for (int i = 0; i < fArr.GetLast(); i++)
   if ((track = (TTrack*)fArr[i])) // or fArr.At(i)
      track->Draw();
```

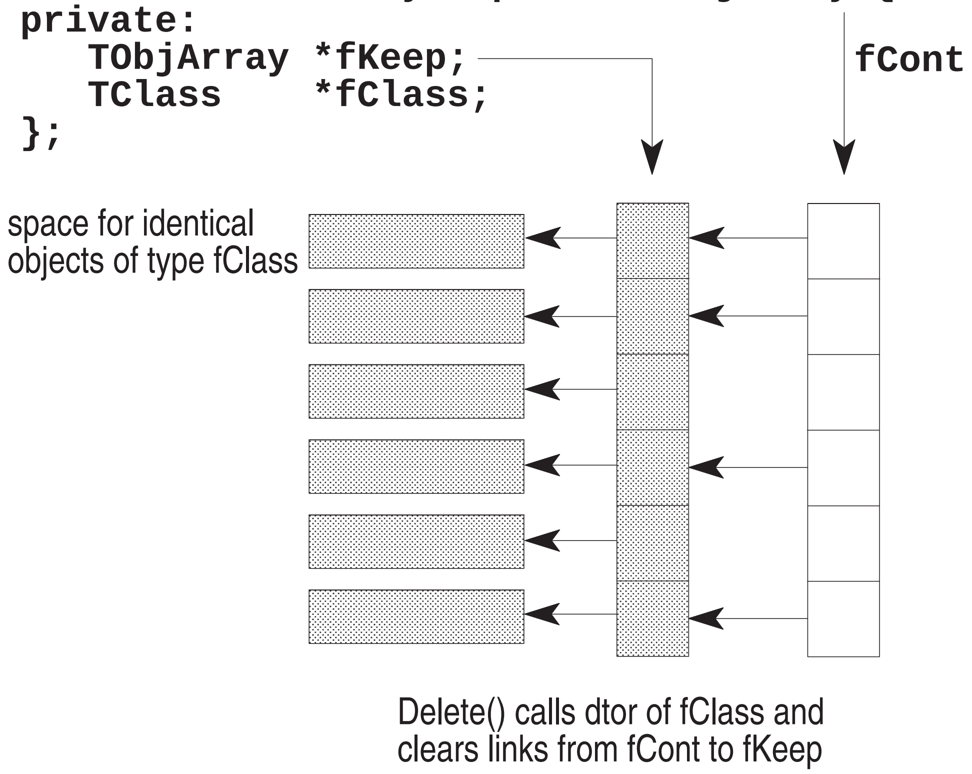
Main features of `TObjArray`: simple, well known array semantics.

Overhead per element: none, except possible oversizing of `fCont`.

TClonesArray - Array of Identical Objects

An array of clone (identical) 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. For the rest this class has the same properties as `TObjArray`.

```
class TClonesArray : public TObjArray {
private:
   TObjArray *fKeep;
   TClass    *fClass;
};
```



The class is specially designed for repetitive data analysis tasks, where in a loop many times the same objects are created and deleted.

Theory of the `TClonesArray`

To reduce the very large number of new and delete calls in large loops like this (O(100000) x O(10000) times new/delete):

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

One better uses a `TClonesArray` which reduces the number of new/delete calls to only O(10000):

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

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

For the other collections see the reference guide on the web and the test program `$ROOTSYS/test/tcollex.cxx`.

Intermezzo: ROOT Collections Compared to Zebra

The Zebra MZ package supports lists by including in each bank (object) a previous and next pointer. The problem with this scheme is that each bank can only be member of one list at a time.

Using the ROOT collection classes objects can be in a large number of collections at the same time.

The same goes for iterators. Using iterators you can make different traversals of a collection at the same time. This would not be possible if each collection would contain iterator state information, like a pointer to the current element.

The ROOT Histogram Classes

ROOT contains a rich set of histogram classes.

The histogram classes are named `TH`**dp** where **d** (dimension) can be `1`, `2` or `3`. The precision **p** can be `C` (for char), `S` (for short), `F` (for float) or `D` (for double). For all dimensions, both fixed and variable bin size options are supported.

For example, **TH1F** is a class for 1-D histograms where a float is used to increment the sum of weights for each channel. That means that you can create histograms with 1, 2, 4 or 8 bytes per channel. **TH2D** is a 2-D histogram with a double word to store the sum of weights.

In addition, a special class **TProfile** handles profile histograms.

All these histogram classes derive from the base class **TH1**. Therefore, most functions described below can be found in the documentation of `TH1`. The `TH1` class, in turn, derives from **TNamed**. Histograms are identified by a name and they have a title. `TH1` also derives (via multiple inheritance) from the standard ROOT attribute classes:

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

```
class TH1 : public TNamed, public TAttLine,
            public TAttFill, public TAttMarker { };
```

The `TH1` class defines an extensive set of histogram functions, including functions for: adding, multiplying, scaling, drawing, setting, getting, fitting, etc.

The ROOT Histogram Classes (contd.)

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

The **`TH1S`** class has a similar limit at 65536.

`TH1F` is the most frequently used histogram class (equivalent to `HBOOK1`). It uses a 32 bits floating point to store the sum of weights per channel.

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

The **`TProfile`** class derives from `TH1D`.

The `TH2`**`p`** classes derive from the corresponding `TH1`**`p`** classes and also from the auxiliary class `TH2`.

The `TH3`**`p`** classes derive from the corresponding `TH1`**`p`** classes and also from the auxiliary class `TH3`.

Creating Histograms

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

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

where `nbins` is the number of bins and `xlow` is the low-edge of the first bin and `xup` is the upper-edge of the last bin, so:

```
TH1F("hfix", "hf title", 3, 0, 3)
```

defines a histogram with 3 bins ranging from 0 to 3 (bin 1 = 0-1, bin 2 = 1-2, bin 3 = 2-3).

A variable bin size histogram is created with:

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

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

2-D histograms are created in the same way (4 different constructors for each precision) with fixed/variable bins along x and/or y.

3-D histograms, currently, have either all fixed bins or all variable bins.

When a histogram is created, it is added to the list of objects in memory in the *current directory* (we will soon come back to the concept of directories in ROOT).

Filling Histograms

Histograms of all types are filled using the appropriate **`Fill()`** function:

- **`Fill(x)`** increment by 1 the bin corresponding to `x`.
- **`Fill(x,w)`** increment by `w` the bin corresponding to `x`.
- **`Fill(x,y)`** increment by 1 the cell corresponding to `x` and `y`.
- **`Fill(x,y,w)`** increment by `w` the cell corresponding to `x` and `y`.
- **`Fill(x,y,z)`**, etc. for 3-D histograms.

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

In case, the function **`TH1::Sumw2()`** has been called before filling the histogram, the sum of squares of weights is also incremented for each channel or cell.

Compared to HBOOK/PAW, the `Fill()` function always increments the histogram data to later compute the mean and the standard deviation.

Drawing Histograms

All histograms can be drawn using the **`Draw("`*`option`*`")`** member function. Many different draw options are supported.

The following options are supported by all histograms:

- `"SAME"` Superimpose on previous picture in the same pad
- `"CYL"` Use Cylindrical coordinates
- `"POL"` Use Polar coordinates
- `"SPH"` Use Spherical coordinates
- `"PSR"` Use PseudoRapidity/Phi coordinates
- `"LEGO"` Draw a lego plot with hidden line removal
- `"LEGO1"` Draw a lego plot with hidden surface removal
- `"LEGO2"` Draw a lego plot using colors to show the cell contents
- `"SURF"` Draw a surface plot with hidden line removal
- `"SURF1"` Draw a surface plot with hidden surface removal
- `"SURF2"` Draw a surface plot using colors to show the cell contents
- `"SURF3"` Same as SURF with in addition a contour view drawn on the top
- `"SURF4"` Draw a surface using Gouraud shading

Drawing Histograms (contd.)

The following options are supported by 1-D histograms:

- `"A"` Do not draw the axis labels and tick marks
- `"B"` Bar chart option
- `"C"` Draw a smooth curve through the histogram bins
- `"E"` Draw error bars
- `"E0"` Draw error bars including bins with 0 contents
- `"E1"` Draw error bars with perpendicular lines at the edges
- `"E2"` Draw error bars with rectangles
- `"E3"` Draw a fill area through the end points of the vertical error bars
- `"E4"` Draw a smoothed filled area through the end points of the error bars
- `"L"` Draw a line through the bin contents
- `"P"` Draw current marker at each bin
- `"*"` Draw a * at each bin

Drawing Histograms (contd.)

The following options are supported by 2-D histograms:

- `"ARR"` Arrow mode, shows gradient between adjacent cells
- `"BOX"` A box is drawn for each cell with surface proportional to contents
- `"COL"` A box is drawn for each cell with a color scale varying with contents
- `"CONT"` Draw a contour plot (same as `CONT3`)
- `"CONT0"` Draw a contour plot using colors to distinguish contours
- `"CONT1"` Draw a contour plot using line styles to distinguish contours
- `"CONT2"` Draw a contour plot using the same line style for all contours
- `"CONT3"` Draw a contour plot using fill area colors
- `"FB"` With `LEGO` or `SURFACE`, suppress the Front-Box
- `"BB"` With `LEGO` or `SURFACE`, suppress the Back-Box
- `"SCAT"` Draw a scatter-plot (default)

All options are case insensitive.

Have a look at the file `$ROOTSYS/tutorials/h1draw.C` to see how the different draw options can be used. Also the `TH1` reference web page shows many examples of draw options.

Once a histogram has been drawn, you can modify its attributes and/or drawing options by using the *contextmenu* item `DrawPanel`.

Writing Histograms

Like any ROOT object, histograms can be written to a file (made persistent). For example:

```
hist->Write();
```

where `hist` is a pointer to a `TH1` derived object.

When a histogram is created, it is added in the list of objects of the current directory in memory. If no file is open at the time of its creation, the histogram is added to the default list of objects in memory. If a file exists, the histogram is added to the list of in memory objects associated to the file. To write the full list of associated memory objects to the file, do:

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

where `file` is the pointer to the file (`TFile`) object.

More on files and how to make (your own) objects persistent will follow soon.

Reading Histograms

Assume you have opened a file using, for example:

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

you can do

```
f.ls()
```

to see the list of objects in this file. In an interactive session, you can read directly a histogram in memory, by just typing its name, or:

```
hist1->Draw()
```

ROOT will automatically load the object `hist1` in memory and create a pointer to it with the same name. Note that this is a ROOT extension to C++ and only works while in an interactive ROOT session.

The recommended way (working both in compiled and interpreted mode) is to do:

```
TH1F *hist1 = (TH1F*)f.Get("hist1");
```

In this case, it is your responsibility to apply the correct cast.

You can also use the browser (see **`TBrowser`**). Double-clicking on a histogram, loads the histogram in memory and draws it.

A histogram is read in memory in the list of memory objects associated to the current directory. This allows for histograms with the same name in different files.

Histogram Operators

For each histogram class the following operators are overloaded:

- `TH`**dp**`& operator=(TH`**dp**`&)`
- `TH`**dp** `operator+(TH`**dp**`&, TH`**dp**`&)`
- `TH`**dp** `operator-(TH`**dp**`&, TH`**dp**`&)`
- `TH`**dp** `operator*(TH`**dp**`&, TH`**dp**`&)`
- `TH`**dp** `operator/(TH`**dp**`&, TH`**dp**`&)`
- `TH`**dp** `operator*(TH`**dp**`&, Float_t)`

Where **dp** are dimension and precision.

Using these operators you can write expressions like:

```
TH1F h = h1*2.5 + h2*8;
TH1F hf;
hf = h/h1;
```

Note that this works only with references and not pointers. Pointers need to be dereferenced:

```
TH1S hs = (*hs1) + (*hs2);
hs.Draw();
```

Intermezzo: Converting from HBOOK to ROOT

Using the program **h2root** you can convert your current PAW/HBOOK files to ROOT files. Simply do:

`h2root histos.hbook`

This will create the file **histos.root**. All HBOOK histograms, profiles and Ntuples (RWN and CWN) in all directories will be converted to `TH`**d**`F`'s, `TProfile`'s and `TNuple`'s. The directory structure will be conserved.

Ntuples variable names are capitalized. For example, `TRACK` becomes `Track` and `PX` becomes `Px`.

Thanks to `h2root` you can quickly and easily migrate from PAW to ROOT.

Fitting in ROOT

Fitting in ROOT is based on the **TMinuit** class. The `TMinuit` class is a re-write in C++ of the well known FORTRAN version.

You can either directly use the `TMinuit` class or use specialized functions provided in the `TH1` histogram class or the graph classes `TGraph` and `TGraphErrors`.

The `TMinuit` class acts on a multiparameter fit function **FCN**. In the ROOT implementation, the function `FCN` is defined via the **TMinuit::SetFCN()** member function when a **TH1::Fit()** command is invoked. The value of `FCN` will in general depend on one or more variable parameters.

To take a simple example, in case of `TH1`**p** histograms the `Fit()` function defines the `TMinuit` fitting function as being `H1FitChisquare()`. `H1FitChisquare()` calculates the chi-square between the user fitting function (gaussian, polynomial, user defined, etc.) and the data for given values of the parameters. It is the task of `TMinuit` to find those values of the parameters which give the lowest value of chi-square.

Fitting 1-D Histograms

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

With pre-defined functions

The following functions are automatically created when any fitting function is invoked:

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

For example, the following command will fit a histogram object `hist` with a gaussian:

```
hist->Fit("gaus");
```

By default, the fitting function object will be added to the histogram object and it will be drawn on top of the histogram. For pre-defined functions, there is no need to set initial values for the parameters. ROOT will do it automatically for you.

Fitting 1-D Histograms (contd.)

With user-defined functions

You can fit using a **TF1** function object. The TF1 function may be created using known expressions, like: sin, cos, exp, etc. (see TFormula for reference). For example, the following command creates a function called "myfit" with 3 parameters in the range between 0 and 2:

```
TF1 *myfit = new TF1("myfit",
             "[0]*sin(x) + [1]*exp(-[2]*x)", 0, 2, 3);
```

Next, we can set parameter names (**optional**) and parameter start values (**mandatory**):

```
myfit->SetParName(0, "c0");
myfit->SetParName(1, "c1");
myfit->SetParName(2, "slope");
myfit->SetParameter(0, 1);
myfit->SetParameter(1, 0.05);
myfit->SetParameter(2, 0.2);
```

We are now ready to fit:

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

You can also create your own C++ fitting function.

This function must have 2 parameters:

1 **Double_t *x**: a pointer to the variables array. This array must be a 1-D array in case of a 1-D histogram, a 2-D array for a 2-D histogram, etc.

2 **Double_t *par**: a pointer to the parameters array. par will contain the current values of parameters when it is called by the FCN function.

Fitting 1-D Histograms (contd.)

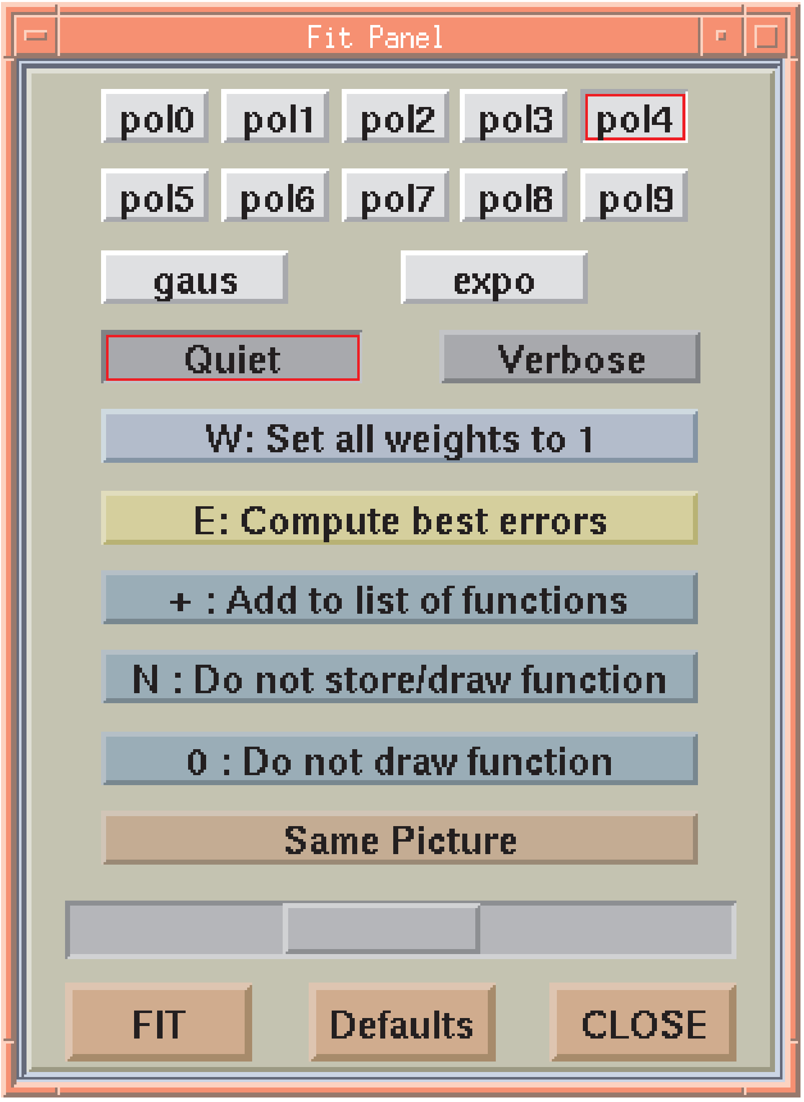
The following macro `fitexample.C` illustrates how to fit a 1-D histogram stored in a ROOT file with a user-defined function:

```
//_________macro fitexample.C___________
Double_t fitf(Double_t *x, Double_t *par)
{
   Double_t arg = 0;
   if (par[2]) arg = (x[0] - par[1])/par[2];

   Double_t fitval = par[0]*TMath::Exp(-0.5*arg*arg);
   return fitval;
}

void fitexample()
{
   TFile *f = new TFile("hsimple.root");

   TH1F *hpx = (TH1F*)f->Get("hpx");

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

Fitting 1-D Histograms (contd.)

Fitting a sub-range of the histogram bins

By default, `TH1::Fit()` will fit the function on the defined histogram range. You can specify the option "**r**" in the second parameter to restrict the fit to the range specified in the `TF1` object. For more complete examples, see the tutorials ''**A simple fitting example**'' and ''**Fitting histogram sub-ranges**''.



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

All these options are available via the histogram `FitPanel`. Accessable via the histogram contextmenu.