Adding Your Own Classes to ROOT

If you want to integrate your classes into the ROOT system, to enjoy features like, extensive RTTI and ROOT object I/O and inspection, you have to add the following line to your class header files:

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

For example in `TLine.h` we have:

```
ClassDef(TLine,1)  //A line segment
```

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

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

```
ClassImp(ClassName)
```

For example in `Line.cxx`:

```
ClassImp(TLine)
```

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

The **`ClassDef`** and **`ClassImp`** macros are necessary to link your classes to the dictionary generated by CINT.

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

Intermezzo: The Default Constructor

ROOT object I/O requires every class to have a default constructor. This default constructor is called whenever an object is being read from a ROOT database. Be sure that you don't allocate any space for embedded pointer objects in the default constructor. This space will be lost (memory leak) while reading in the object. For example:

```
class T49Event : public TObject {
private:
   Int_t        fId;
   TCollection *fTracks;
   ...
   ...
public:
   // Error space for TList pointer will be lost
   T49Event() { fId = 0; fTrack = new TList; }
   // Correct default initialization of pointer
   T49Event() { fId = 0; fTrack = 0; }
   ...
   ...
};
```

The memory will be lost because during reading of the object the pointer will be set to the object it was pointing to at the time the object was written.

Create the **fTrack** list when you need it, e.g. when you start filling the list or in a **not-default** constructor.

```
...
if (!fTrack) fTrack = new TList;
...
```

The CINT Dictionary Generator

In the following example we walk through the steps necessary to generate a dictionary and I/O and inspect member functions.

Let start with an TEvent class which contains a collection of TTracks:

```
#ifndef __TEvent__
#define __TEvent__

#include "TObject.h"

class TCollection;
class TTrack;


class TEvent : public TObject {

private:
   Int_t        fId;          //event sequential id
   Float_t      fTotalMom;    //total momentum
   TCollection *fTracks;      //collection of tracks

public:
   TEvent() { fId = 0; fTracks = 0; }
   TEvent(Int_t id);
   ~TEvent();

   void    AddTrack(TTrack *t);
   Int_t   GetId() const { return fId; }
   Int_t   GetNoTracks() const;
   void    Print(Option_t *opt="");
   Float_t TotalMomentum();

   ClassDef(TEvent,1)  //Simple event class
};
```

The CINT Dictionary Generator (contd.)

And the TTRack header:

```
#ifndef __TTrack__
#define __TTrack__

#include "TObject.h"

class TEvent;


class TTrack : public TObject {

private:
   Int_t    fId;         //track sequential id
   TEvent  *fEvent;      //event to which track belongs
   Float_t  fPx;         //x part of track momentum
   Float_t  fPy;         //y part of track momentum
   Float_t  fPz;         //z part of track momentum

public:
   TTrack() { fId = 0; fEvent = 0; fPx = fPy = fPz = 0; }
   TTrack(Int_t id, Event *ev, Float_t px, Float_t py,
                               Float_t pz);

   Float_t  Momentum() const;
   TEvent  *GetEvent() const { return fEvent; }
   void     Print(Option_t *opt="");

   ClassDef(TTrack,1)  //Simple track class
};

#endif
```

The CINT Dictionary Generator (contd.)

The things to notice in these header files are:

- The usage of the `ClassDef` macro
- The default constructors of the `TEvent` and `TTrack` classes
- The usage of comments to describe the data members and the comment after the `ClassDef` macro to describe the class.

The intended usage of these classes is that one creates and event object with a certain id and then add tracks to the event. As one can see the track objects contain a pointer to the event to which they belong. This to show that the I/O system will correctly handle circular references.

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

```
#include <iostream.h>

#include "TOrdCollection.h"
#include "TEvent.h"
#include "TTrack.h"

ClassImp(TEvent)

...
...
```

and Track.cxx:

```
#include <iostream.h>

#include "TMath.h"
#include "Track.h"
#include "Event.h"

ClassImp(TTrack)

...
...
```

The CINT Dictionary Generator (contd.)

Now using **rootcint** we can generate the dictionary file:

rootcint eventdict.cxx -c TEvent.h TTrack.h

Looking in the file **eventdict.C** we can see, besides the many member function calling stubs (used internally by the interpreter), the **Streamer()** and **ShowMembers()** functions for the two classes. `Streamer()` is used to stream an object to/from a `TBuffer` and `ShowMembers()` is used by the `Dump()` and `Inspect()` methods of `TObject`. Here is the `TEvent::Streamer()` method:

```
//________________________________________________________
void TEvent::Streamer(TBuffer &R__b)
{
   // Stream an object of class TEvent.

   if (R__b.IsReading()) {
      Version_t R__v = R__b.ReadVersion();
      TObject::Streamer(R__b);
      R__b >> fId;
      R__b >> fTotalMom;
      R__b >> fTracks;
   } else {
      R__b.WriteVersion(TEvent::IsA());
      TObject::Streamer(R__b);
      R__b << fId;
      R__b << fTotalMom;
      R__b << fTracks;
   }
}
```

The **TBuffer** class overloads the `operator<<()` and `operator>>()` for all basic types and for pointers to objects. These operators write and read from the buffer and take care of any needed byte swapping to make the buffer machine independent. During writing the `TBuffer` keeps track of the objects that have been written and multiple references to the same object are replaced by an index. Also the object's class information is stored.

The CINT Dictionary Generator (contd.)

Concerning `Streamer()`:

- It does not know what to do with pointers to basic types
- It does not know what to do with non-`TObject` derived objects (or struct's)

Both these cases need manual intervention. Cut and paste the generated `Streamer()` in the class' source file and modify as needed (e.g. add counter for array of basic types) and disable the generation of the `Streamer()` using the **`LinkDef.h`** file (see **`rootcint -?`**) for next runs of `rootcint`.

To exclude a data member from the `Streamer()` add **!** in comment field, e.g.:

```
Int_t fTempVal;   //! temp state value
```

To prevent generation of `Streamer()`, in case you don't want to do I/O (and not to prevent the generation of a `Streamer()` because you already have a customized version), do:

```
ClassDef(TEvent,0)
```

The CINT Dictionary Generator (contd.)

And here is the `TEvent::ShowMembers()`:

```
//______________________________________________
void TEvent::ShowMembers(TMemberInspector &R__insp,
                         char *R__parent)
{
   // Inspect the data members of an object of
   // class TEvent.

   TClass *R__cl  = TEvent::IsA();
   Int_t   R__ncp = strlen(R__parent);
   R__insp.Inspect(R__cl, R__parent, "fId", &fId);
   R__insp.Inspect(R__cl, R__parent, "fTotalMom",
                   &fTotalMom);
   R__insp.Inspect(R__cl, R__parent, "*fTracks",
                   &fTracks);
   TObject::ShowMembers(R__insp, R__parent);
}
```

The `ShowMembers()` method gives access, via a `TMemberInspector` object, to an object's data member names and their matching addresses. This allows for complete introspection of every ROOT object during run time (via `TObject::Dump()` and `TObject::Inspect()`).

As can be seen both `Streamer()` and `ShowMembers()` call correctly up the hierarchy of base classes.

Extending ROOT with Shared Libraries

We can now compile the `TEvent` and `TTrack` classes and the `eventdict.cxx` dictionary file and link them together into a single shared library, **event.so** (see `$ROOTSYS/test/Makefile` on how to create a shared library for your platform).

Loading a shared library in a running ROOT session is trivial, first start `root`, then type:

```
root [0] gSystem.Load("event.so")
(int)0
```

A return value of `0` means that the load was successful. Now lets verify that the classes `TEvent` and `TTrack` can be seen by the system:

```
root [1] .class TEvent
===================================================
class TEvent //Simple event class
...
...
root [2] gClassTable.Print()
Defined classes
class                                   version  initialized
========================================================
TEvent                                      1        No
TTrack                                      1        No
...
...
```

When loading a shared library using `gSystem.Load()` the system will look for the library in the **Root.DynamicPath** search path as specified in your `.rootrc` file.

So, instead of re-linking the root executable with your libraries you should use shared libraries to dynamically extend the system. Typically, you put the `gSystem.Load()` command(s) in your `rootlogon.C` file.

Writing Objects to a ROOT Database

Objects inheriting from the class `TObject` and having a valid `Streamer()` method can be made persistent (i.e. written to a file). Before writing an object to a file, you must first create a **NEW** file or open an existing file in **UPDATE** or **RECREATE** mode. For example:

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

Objects in a file are identified by a key (see class `TKey`). A key is an object with all the necessary information to locate an object stored in a file (its name, title, size, position and a few other parameters). A file has a directory consisting of the list of keys. A key is automatically created by an operation such as **obj->Write("*keyname*")**. **Write()** is a member function of the `TObject` class and performs the following operations:

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

When your object derives from **TNamed**, you may omit the parameter "***keyname***". For example, for a `TH1F` object, you can write: **obj->Write()**. The key created in this case will get the name from the `TNamed` object. If you write a key with a name already existing in the file, a new cycle (a la VMS) is created. You can use `TFile::ls()` or `TFile::Map()` to see the list of all keys (i.e. objects) and records in a file.

Use `TDirectory::pwd()` to see the current directory. To make subdirectories in a file or directory create `TDirectory` objects:

```
TDirectory *dir = new TDirectory("dirname", "title");
dir->cd();
```

The current file and directory can be accessed via the globals **gFile** and **gDirectory** (e.g. `gDirectory->pwd()`).

Reading Objects From a ROOT Database

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

Via **`TFile::Get()`** or **`TDirectory::Get()`** if you are in a sub-directory. Assume an existing file `f` (created by `TFile f("myfile.root")`). You can list all the keys (persistent objects in the file) via `f.ls()`. To read an object in memory (for example a TH1F object), you can do:

```
TH1F *hist = (TH1F*)f.Get("hist_name");
```

`f.Get()` will create an object of class TH1F using the following sequence of operations:

- Find the key (e.g. hist_name) in the list of keys
- Create a `TBuffer` object
- Read the buffer from the file
- Create an empty object by calling the default constructor for the class referenced in the `TKey`
- Call the `Streamer()` function for this new object

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

This method via `TDirectory::Get()` is interesting to return an object by its name in complete random access.

Reading Objects From a ROOT Database (contd.)

In case of a very long list of objects to be processed sequentially (this could be a list of events), it is simpler and more efficient to scan the list of keys directly. The example below illustrates at the same time the use of an iterator to loop on all keys of a file:

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

Extra: The TWebFile

Make your ROOT files available to colleagues via the WWW. Use a **TWebFile**. A `TWebFile` is a read-only `TFile` that allows the reading of a ROOT database via a standard (slightly modified) Apache webserver. For example:

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

Intermezzo: Navigating in a ROOT Database

A ROOT database can have directories and sub-directories just like a file system. Using these directories you can group and order your objects in a logical hierarchy.

You navigate in the database using the global pointer, **`gDirectory`**. This pointer always points to the current directory (`TDirectory`). On connecting a file (`TFile`) the `gDirectory` points to the top directory of the new file. Using the `TDirectory` methods:

- `mkdir(const char *name, cont char *title="")`
- `cd(const char *dir)`
- `ls()`
- `pwd()`

you can create a new directory, change directory, list objects in a directory and print the path of the current directory.

For example:

```
root [0] TFile f("db.root","new")
root [1] f.mkdir("histos")
root [2] f.mkdir("ntuples")
root [3] f.cd("histos")
root [4] TH1F *h = new TH1F("hist1","hist1", 100, 0, 100)
root [5] h.Write()
root [6] gDirectory.ls()
TDirectory*            histos  histos
 OBJ: TH1F      hist1   histogram 1 : 0
 KEY: TH1F      hist1;1 histogram 1
root [7] gDirectory.pwd()
db.root:/histos
root [8] gDirectory.cd("..")
root [9] gDirectory.pwd()
db.root:/
root [10] f.Write()
TFile Writing Name=db.root Title=
root [11] f.Close()
```

Storing Many, Many Identical Objects