

ROOT Course

Fons Rademakers

GSI Darmstadt

Reference material:

- The ROOT web site: <http://root.cern.ch/>
- Digest of roottalk mailing list:
<http://root.cern.ch/root/roottalk/>

Tutorials come with the ROOT distribution, see directories:

- \$ROOTSYS/test
- \$ROOTSYS/tutorials

Basic understanding of OOP and C++ assumed

Prehistory

In the beginning there was PAW

- HBOOK
- ZEBRA
- KUIP
- COMIS
- SIGMA

Mini/Micro-DST analysis was done using Ntuples (RWN and CWN)

- Ntuples are basically simple tables
- Only basic types (RWN only floats)
- No data structures
- No cross reference between Ntuples
- Successful because simple and efficient

Dead-end

- No way to grow to more complex data structures
- Difficult to extend
- Expensive to maintain
- Written in soon to be obsolete language -:)

Main Goals for New System

Being able to support full data analysis chain

- Raw data, DSTs, mini-DSTs, micro-DSTs

Being able to handle complex structures

- Complete objects
- Object hierarchies

Support at least the PAW data analysis functionality

- Histogramming
- Fitting
- Visualization

Only one language

- C++

Better maintainable

- Use OOP

Make the system extensible

- Use OO framework technology

Advantages of Object-Oriented Programming

Object-Oriented programming has been around for at least 20 years. However, it has never been a serious option for scientific computing due to the large performance overhead.

Only with the introduction of C++, which inherited a lot of its syntax and efficiency from the popular C language, was there a viable alternative.

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

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

Frameworks

A *framework* is a collection of cooperating classes that make up a reusable design solution for a given problem domain.

There are three main differences between frameworks and class libraries:

Behaviour versus protocol. Class libraries are essentially collections of behaviours that you can call when you want those individual behaviours in your program. A framework, on the other hand, provides not only behaviour but also the protocol or set of rules that govern the ways in which behaviours can be combined.

Don't call us, we'll call you. With a class library, the code the programmer writes instantiates objects and calls their member functions. With a framework a programmer writes code that overrides and is called by the framework. The framework manages the flow of control among its objects. This relationship is expressed by the principle: "Don't call us, we'll call you".

Implementation versus design. With class libraries programmers reuse only implementations, whereas with frameworks they reuse design. A framework embodies the way a family of related classes work together.

Calling API Versus Subclassing API

Frameworks can be thought of as having two Application Programmer Interfaces (APIs): a calling API, which resembles a class library, and a subclassing API, which is used for overriding framework member functions.

The calling/subclassing distinction describes the mechanism by which functions are invoked: call or be called.

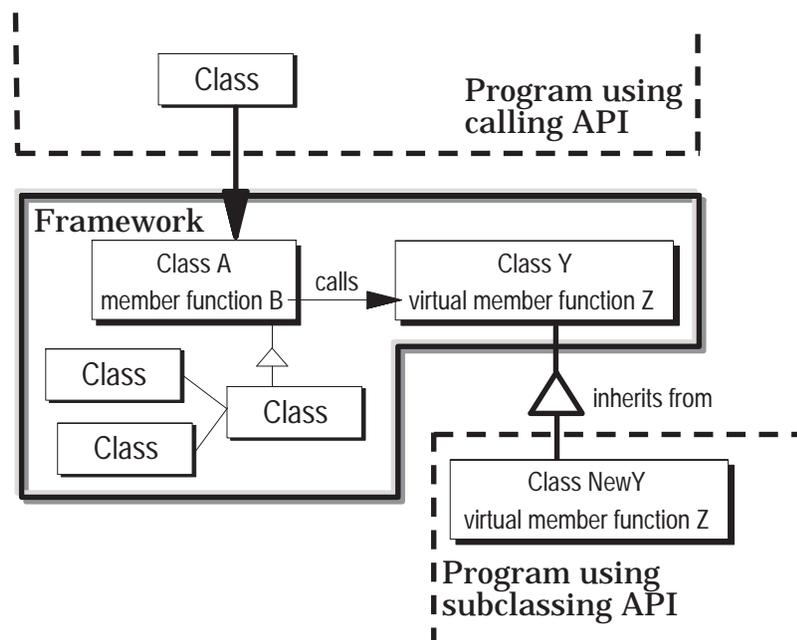


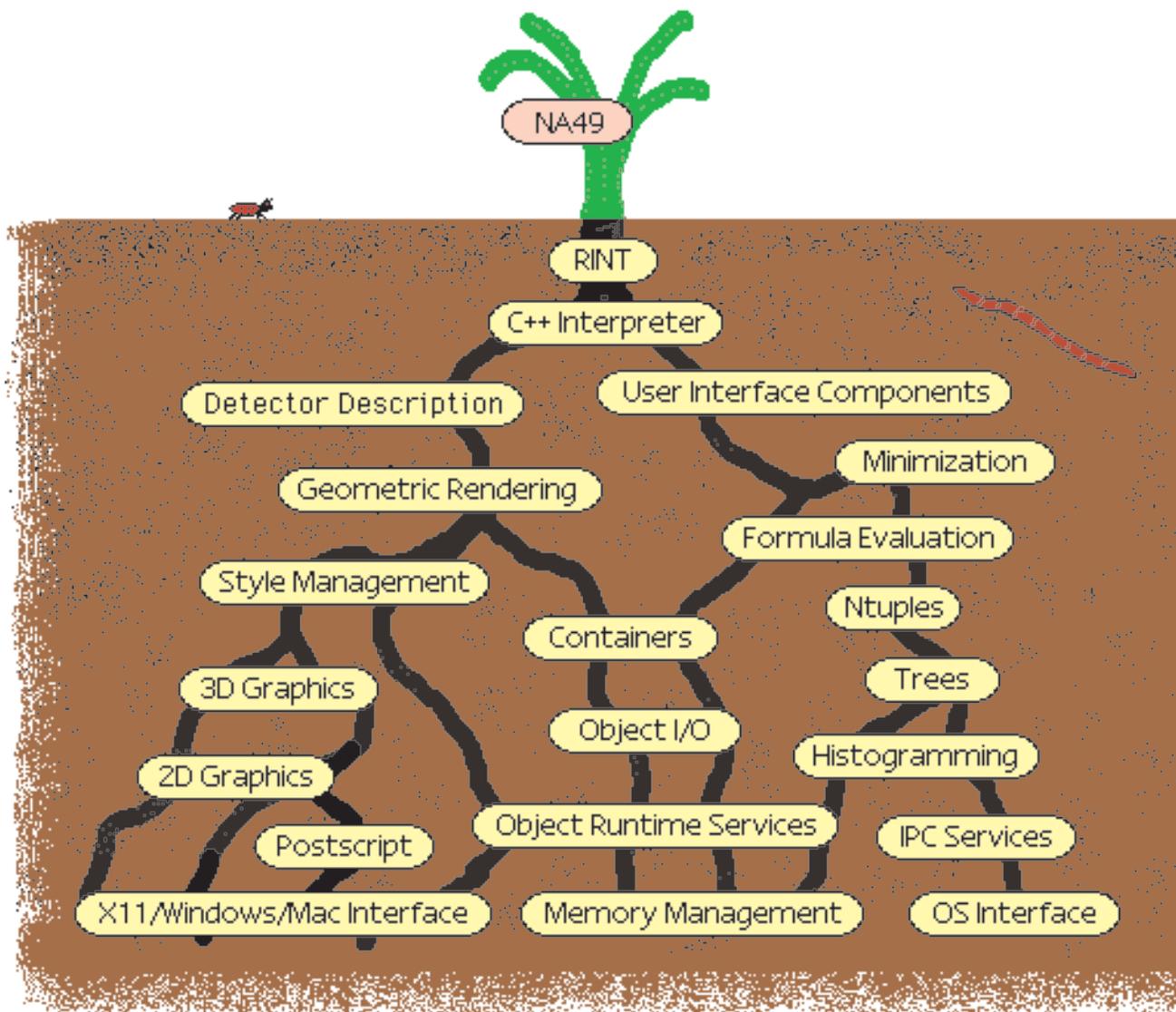
Figure 2 Calling API versus subclassing API

Advantages of Frameworks

The benefits of frameworks can be summarized as follows:

- **Less code to write.** Much of the program's design and structure, as well as its code, already exist in the framework.
- **More reliable and robust code.** Code inherited from a framework has already been tested and integrated with the rest of the framework.
- **More consistent and modular code.** Code reuse provides consistency and common capabilities between programs, no matter who writes them. Frameworks also make it easier to break programs into smaller pieces.
- **More focus on areas of expertise.** Users can concentrate on their particular problem domain. They don't have to be experts at writing user interfaces, graphics, or networking to use the frameworks that provide those services.

Schematic View of the ROOT System



The ROOT Class Categories

The ROOT system consists of the following class categories and frameworks:

- Basic Classes
- Container Classes
- Histogram and Minimization Classes
- Tree and Ntuple Classes
- 2D Graphics Classes
- 3D Graphics and Detector Geometry Classes
- Graphical User Interface (Motif and Win32) Classes
- Meta Classes + Interface to the CINT C++ Interpreter
- Operating System Interface Classes
- Networking Classes
- Documentation Classes
- Interactive Interface
- PROOF Server

What Do I Mean With “ROOT”

ROOT is:

- 1 A set of OO frameworks that a programmer can use to build applications. The frameworks are available as a set of header files and libraries. Use and/or inherit from the ROOT classes and link with the libraries to build your application. ROOT is not an "all or nothing" proposition, you can use only the parts you need, e.g. histogramming, graphics, etc.
- 2 An interactive stand-alone application, `root` (or `root.exe`), which gives access to all ROOT classes via a command line interpreter or GUI. The `root` application can be extended at run time by loading shared libraries of user code.

The command line interpreter is a C++ interpreter:

```
$ root
root[0] printf("Hello World\n");
Hello World
root [1]
```

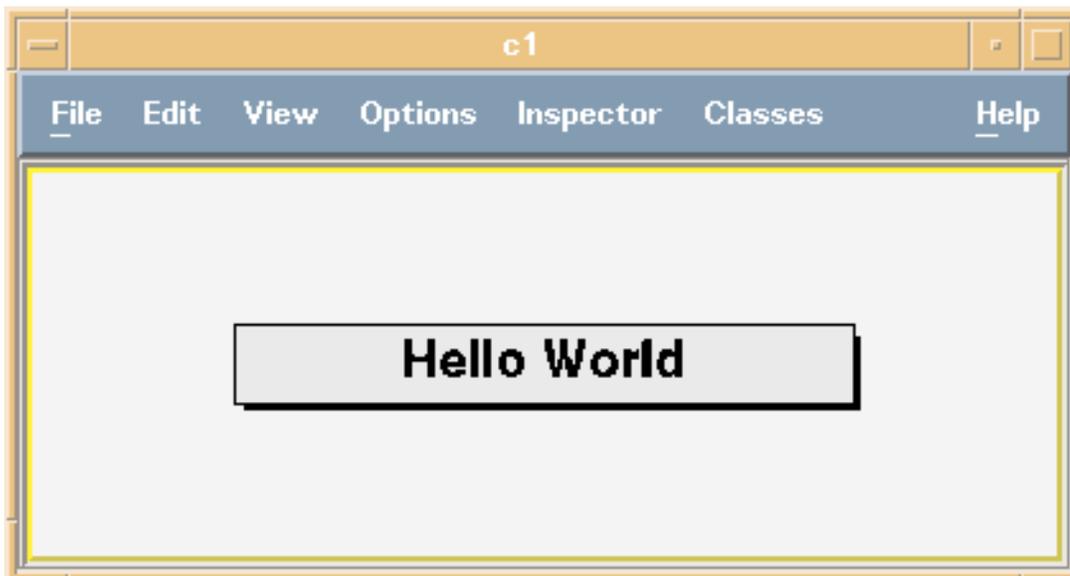
Running the Interactive ROOT Program

The same Hello World with some graphics...

Start the interactive version of ROOT and execute the following two commands:

```
$ root
root[0] TPaveLabel hello(0.2,0.4,0.8,0.6,"Hello World")
root[1] hello.Draw()
```

The first command creates an object of class `TPaveLabel`. The second command will draw the `pavelabel` object (this will automatically create a graphics canvas).



You can now use the mouse to:

- Move the mouse on the pave and press the left button. Keeping the left button pressed, you can move the pave in the canvas.
- Point to one of the corners of the pave and grow or shrink the pave.
- Move the mouse on the pave and press the right button. You get a context pop-up menu showing a list of possible pave member functions.

ROOT Command Line Options

The root executable support the following command line options:

```
(hproot) [771] root -?  
Usage: root [-b] [-n] [-q] [file1.C ... fileN.C]  
Options:  
  -b : run in batch mode without graphics  
  -n : do not execute logon and logoff macros as  
       specified in .rootrc  
  -q : exit after processing command line macro files
```

At start-up root looks for a .rootrc file in the following order:

- ./rootrc (local)
- \$HOME/.rootrc (user)
- \$ROOTSYS/.rootrc (global)

The options are merged, with precedence local, user, global
(do: gEnv->Print() to see current settings)

The .rootrc file typically looks like:

```
# Path used by dynamic loader to find shared libraries  
Unix.*.Root.DynamicPath:  .:~/rootlibs:$ROOTSYS/lib  
Unix.*.Root.MacroPath:    .:~/rootmacros:$ROOTSYS/macros  
  
# Activate memory statistics  
Rint.Root.MemStat:        1  
Rint.Load:                rootalias.C  
Rint.Logon:               rootlogon.C  
Rint.Logoff:              rootlogoff.C  
  
Rint.Canvas.MoveOpaque:  false  
Rint.Canvas.HighLightColor: 5
```

The rootlogon.C and rootlogoff.C files are simple macro files that will be loaded and executed at start-up and shutdown of root. The rootalias.C file will be loaded but not executed. It typically contains small utility functions.

My First ROOT Program

The same Hello World example now as stand-alone application.

```
// Source: $ROOTSYS/test/hworld.cxx
// This small demo shows the traditional "Hello World".
// Its main use is to show how to use ROOT graphics
// and how to enter the eventloop to be able to
// interact with the graphics.

#include "TROOT.h"
#include "TApplication.h"
#include "TCanvas.h"
#include "TPaveLabel.h"

extern void InitGui();
VoidFuncPtr_t initfuncs[] = { InitGui, 0 };

TROOT root("hello","Hello World", initfuncs);

int main(int argc, char **argv)
{
    TApplication theApp("App", &argc, argv);

    TCanvas *c = new TCanvas("Hello", "The Hello Canvas",
                            400, 400);

    TPaveLabel *hello = new TPaveLabel(0.2,0.4,0.8,0.6,
                                        "Hello World");
    hello->Draw();

    // Enter event loop, one can now interact with the
    // objects in the canvas. Select "Exit ROOT" from
    // the canvas "File" menu to exit the program.
    theApp.Run();

    return 0;
}
```

Essential Ingredients of a ROOT Based Program

Every ROOT program *must* contain one TROOT object

- The TROOT object must be created as global object (i.e. before `main()` is called) or it must be the first object created in `main()`. It must exist during the complete life-time of the program.
- The TROOT object can only be created on the stack (protected operator `new`). This guarantees the calling of the TROOT destructor.
- The TROOT object keeps track of all created objects and resources
- The TROOT object is always accessible via the global `gROOT` pointer
- The TROOT class is a singleton

Every ROOT program wanting to interact with the user must contain one TApplication (or TApplication derived) object

- The `TApplication::Run()` method starts the ROOT eventloop
- `Run()` never returns unless its argument is `kTRUE`
- In the eventloop the GUI is active
- Use `TRint` instead of `TApplication` to get also a command line prompt in the eventloop
- The `TApplication` class is a singleton

Intermezzo: The ROOT Program

```
////////////////////////////////////  
//                                                                    //  
// RMain                                                                //  
//                                                                    //  
// Main program used to create ROOT application.                       //  
//                                                                    //  
////////////////////////////////////  
  
#include "TROOT.h"  
#include "TRint.h"  
  
extern void InitGui();  
  
int Error; // needed by Motif on HP-UX  
  
VoidFuncPtr_t initfuncs[] = { InitGui, 0 };  
  
TROOT root("Rint","The ROOT Interactive Interface",  
          initfuncs);  
  
int main(int argc, char **argv)  
{  
    TRint *theApp = new TRint("Rint", &argc, argv, 0, 0);  
  
    theApp->Run();  
  
    delete theApp;  
  
    return(0);  
}
```

Compiling and Linking a ROOT Program

Compiling a ROOT program requires the option:

- `-I$ROOTSYS/include`

Linking requires the option:

- `-L$ROOTSYS/lib`

And the libraries:

- `-lBase -lCint -lClib -lCont -lFunc -lGraf -lGraf3d
-lHist -lHtml -lMeta -lMinuit -lNet -lPostscript
-lProof -lTree -lUnix -lZip`
- `-lGpad -lGX11 -lMotif -lWidgets -lX3d`
(plus Motif and X11 libs)

For a complete example see the file:

`$ROOTSYS/test/Makefile`

Intermezzo: ROOT Coding Convention

We use the following conventions (based on Taligent):

Identifier	Convention	Example
Classes	Begin with T	THashTable
Non-class types	End with _t	Simple_t
Enumeration types	Begin with E	EColorLevel
Data members	Begin with f for field	fViewList
Member functions	Begin with a capital	Draw()
Static variables	Begin with g	gSystem
Static data members	Begin with fg	fgTokenClient
Locals and parameters	Begin with lower case	seed, thePad
Constants	Begin with k	kInitialSize, kRed
Template arguments	Begin with A	AType
Getters and setters	Begin with Get, Set, or Is (boolean)	SetLast(), Get- First(), IsDone()

In any name that contains more than one word, the first word follows the convention for the type of the name, and subsequent words follow with the first letter of each word capitalized, such as TTextBase.

Do not use underscores except for #define symbols.

Don't impose too rigid rules. People will not follow them.

However, the agreed upon rules should be followed religiously otherwise the result will be worse than chaos.

The CINT C++ Interpreter

The CINT C/C++ interpreter has been developed by Masaharu Goto of HP Japan.

CINT has the following main features:

- It implements 95% of ANSI C and 85% of (ANSI) C++.
- It is robust and complete enough to interpret itself (80000 lines of C).
- Interpreter technology allows for very fast prototyping. No more compile/link cycles.
- It is fast. Faster than tcl, perl, python.
- It has good debugging facilities.
- It can be easily interfaced to any C or C++ library by (dynamically) linking the library and library stubs and dictionary (generated by CINT from the library header files).
- It is fully embedded into the ROOT system. It acts as command line and macro processor.
- It provides ROOT with full RTTI of all classes (used a.o. to generate `Streamer()`, `ShowMembers()` and the HTML documentation).
- The command line, macro and programming languages become the same.

The ROOT Command Line Interface

```
(hroot) [199] root
*****
*
*           W E L C O M E   t o   R O O T           *
*
*   Version   1.00/10           25 April 1997       *
*
*   You are welcome to visit our Web site         *
*           http://root.cern.ch                   *
*
*****
```

CINT/ROOT C/C++ Interpreter version 5.13.3, Mar 20 1997
Type ? for help. Commands must be C++ statements.
Enclose multiple statements between { }.

```
root [0] TLine l
root [1] l.Print()
TLine  X1=0.000000 Y1=0.000000 X2=0.000000 Y2=0.000000
root [2] l.SetX1(10)
root [3] l.SetY1(11)
root [4] l.Print()
TLine  X1=10.000000 Y1=11.000000 X2=0.000000 Y2=0.000000
root [5] .g
...
...
0x4038f080 class TLine l , size=40
0x0      protected: Coord_t fX1 //X of 1st point
0x0      protected: Coord_t fY1 //Y of 1st point
0x0      protected: Coord_t fX2 //X of 2nd point
0x0      protected: Coord_t fY2 //Y of 2nd point
0x0      private: static class TClass* fgIsA
```

Here we note:

- Terminating ; not required.
- Emacs style command line editing.
- Raw interpreter commands start with a . (dot).

The ROOT Command Line Interface (contd.)

```
root [6] .class TLine
=====
class TLine //A line segment
  size=0x28
List of base class-----
0x0      public: TObject //Basic ROOT object
0xc      public: TAttLine //Line attributes
List of member variable-----
Defined in TLine
0x0      protected: Coord_t fX1 //X of 1st point
0x0      protected: Coord_t fY1 //Y of 1st point
0x0      protected: Coord_t fX2 //X of 2nd point
0x0      protected: Coord_t fY2 //Y of 2nd point
0x0      private: static class TClass* fgIsA
List of member function-----
Defined in TLine
filename  line:size busy function type and name
(compiled) 0:0      0 public: class TLine TLine(void);
(compiled) 0:0      0 public: Coord_t GetX1(void);
(compiled) 0:0      0 public: Coord_t GetX2(void);
(compiled) 0:0      0 public: Coord_t GetY1(void);
(compiled) 0:0      0 public: Coord_t GetY2(void);
...
...
(compiled) 0:0 public: virtual void SetX1(Coord_t x1);
(compiled) 0:0 public: virtual void SetX2(Coord_t x2);
(compiled) 0:0 public: virtual void SetY1(Coord_t y1);
(compiled) 0:0 public: virtual void SetY2(Coord_t y2);
(compiled) 0:0      0 public: void ~TLine(void);
root [7] l.Print(); > test.log
root [8] l.Dump(); >> test.log
root [9] ?
```

Here we see:

- Use `.class` as quick help and reference.
- Unix like I/O redirection (`;` is required before `>`).
- Use `?` to get help on all “raw” interpreter commands

The ROOT Command Line Interface (contd.)

Now lets execute a multi line command:

```
root [9] {
end with '}' '> TLine l;
end with '}' '> for (int i = 0; i < 5; i++) {
end with '}' '>     l.SetX1(i);
end with '}' '>     l.SetY1(i+1);
end with '}' '>     l.Print();
end with '}' '> }
end with '}' '> }
```

```
TLine  X1=0.000000 Y1=1.000000 X2=0.000000 Y2=0.000000
TLine  X1=1.000000 Y1=2.000000 X2=0.000000 Y2=0.000000
TLine  X1=2.000000 Y1=3.000000 X2=0.000000 Y2=0.000000
TLine  X1=3.000000 Y1=4.000000 X2=0.000000 Y2=0.000000
TLine  X1=4.000000 Y1=5.000000 X2=0.000000 Y2=0.000000
root [10] .q
```

Here we note:

- A multi-line command starts always with a { and ends with a }.
- Every line has to be correctly terminated with a ; (like in “real” C++).
- All objects are created in *global scope*.
- There is no way to back up. Better write a macro.
- Use `.q` to exit root.

For more details see:

<http://root.cern.ch/root/CintInterpreter.html>

The ROOT Macro Processor

ROOT/CINT macro files contain pure C++ code. They can contain a simple sequence of statements like in the multi command line example given above, but also arbitrarily complex class and function definitions.

Lets start with a macro containing a simple list of statements (like the multi command line example given in the previous section). This type of macro must start with a { and end with a }. Assume the file is called `macro1.C`:

```
//-----  
{  
#include <iostream.h>  
  
    cout << " Hello" << endl;  
    float x = 3.;  
    float y = 5.;  
    int    i = 101;  
    cout <<" x = "<<x<<" y = "<<y<<" i = "<<i<< endl;  
}  
//-----
```

To execute the stream of statements in `macro1.C` do:

```
root [1] .x macro1.C
```

This loads the contents of file `macro1.C` and executes all statements in the interpreter's *global scope*.

One can re-execute the statements by re-issuing `".x macro1.C"` (since there is no function entry point).

Macros are searched for in the `Root.MacroPath` as defined in your `.rootrc` file. To check which macro is being executed use:

```
root [2] .which macro1.C  
/home/rdm/root/./macro1.C
```

The ROOT Macro Processor (contd.)

Now copy file `macro1.C` to `macro2.C` and add a function statement. Like this:

```
//-----  
#include <iostream.h>  
  
int main()  
{  
    cout << " Hello" << endl;  
    float x = 3.;  
    float y = 5.;  
    int i = 101;  
    cout <<" x = "<<x<<" y = "<<y<<" i = "<<i<<endl;  
    return 0;  
}  
//-----
```

Notice that no surrounding { and } are required in this case.

To execute function `main()` in `macro2.C` do:

```
root [1] .L macro2.C // load macro in memory  
root [2] main() // execute entry point main  
Hello  
x = 3 y = 5 i = 101  
(int)0  
root [3] main() // execute main() again  
Hello  
x = 3 y = 5 i = 101  
(int)0  
root [4] .func // list all functions known by CINT  
filename line:size busy function type and name  
...  
macro2.C 4:9 0 public: int main();
```

The last command shows that `main()` has been loaded from file `macro2.C`, that the function `main()` starts on line 4 and is 9 lines long. Notice that once a function has been loaded it becomes part of the system just like a compiled function.

The ROOT Macro Processor (contd.)

Now we copy file `macro2.C` to `macro3.C` and change the function name from `main()` to `macro3(int j = 10)`:

```
//-----  
#include <iostream.h>  
  
int macro3(int j = 10)  
{  
    cout << " Hello" << endl;  
    float x = 3.;  
    float y = 5.;  
    int    i = j;  
    cout <<" x = "<<x<<" y = "<<y<<" i = "<<i<<endl;  
    return 0;  
}  
//-----
```

To execute `macro3()` in `macro3.C` type:

```
root [1] .x macro3.C(8)
```

This loads the contents of file `macro3.C` and executes entry point `macro3(8)`. Note that the above only works when the filename (minus extension) and function entry point are both the same. Function `macro3()` can still be executed multiple times:

```
root [2] macro3()  
Hello  
x = 3 y = 5 i = 10  
(int)0  
root [3] macro3(33)  
Hello  
x = 3 y = 5 i = 33  
(int)0
```

The ROOT Macro Processor (contd.)

A Macro Containing a Class Definition

Lets create a small class `TMyClass` and a derived class `TChild`. The virtual `TMyClass::Print()` method is overridden in `TChild`:

```
//-----  
#include <iostream.h>  
  
class TMyClass {  
  
private:  
    float    fX;        //x position in centimeters  
    float    fY;        //y position in centimeters  
  
public:  
    TMyClass() { fX = fY = -1; }  
    virtual void Print() const;  
    void      SetX(float x) { fX = x; }  
    void      SetY(float y) { fY = y; }  
};  
  
void TMyClass::Print() const  
{  
    cout << "fX = " << fX << ", fY = " << fY << endl;  
}  
  
class TChild : public TMyClass {  
public:  
    void Print() const;  
};  
  
void TChild::Print() const  
{  
    cout << "This is TChild::Print()" << endl;  
    TMyClass::Print();  
}  
//-----
```

and save it in file `macro4.C`.

The ROOT Macro Processor (contd.)

To execute macro4.C do:

```
root [0] .L macro4.C
root [1] TMyClass *a = new TChild
root [2] a.Print()
This is TChild::Print()
fX = -1, fY = -1
root [3] a.SetX(10)
root [4] a.SetY(12)
root [5] a.Print()
This is TChild::Print()
fX = 10, fY = 12
root [6] .class TMyClass
=====
class TMyClass
  size=0x8 FILE:macro4.C LINE:3
List of base class-----
List of member variable-----
Defined in TMyClass
0x0      private: float fX
0x4      private: float fY
List of member function-----
Defined in TMyClass
filename      line:size busy function type and name
macro4.C      16:5      0 public: class TMyClass
                                   TMyClass(void);
macro4.C      22:4      0 public: void Print(void);
macro4.C      12:1      0 public: void SetX(float x);
macro4.C      13:1      0 public: void SetY(float y);
root [7] .q
```

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

Current limitation:

- Classes defined in a macro can not inherit from TObject. Currently the interpreter can not patch the virtual table of compiled objects to reference interpreted objects.

Variable Scope and Resetting the Interpreter Environment

All objects created on the command line are in the interpreter's global scope. Also all *non-function* macros create objects in the global scope.

Therefore:

```
root [0] TLine l
root [1] TPolyLine l
Error: l already declared as different type. ~TPolyLine()
called
FILE:/tmp/01156baa LINE:1
root [2] TLine *l1 = new TLine
root [3] TPolyLine *l1 = new TPolyLine
Error: l1 already declared as different type
FILE:/tmp/01156daa LINE:1
```

To reset the global scope use the function `gROOT.Reset()`

This clears the global scope to the status just before executing the previous macro (not including any logon macros). Therefore non-function macros often start with the statement `gROOT.Reset()`.

When clearing the global scope the destructors of objects are called (as expected). Objects created on the heap (via `new`) are not deleted (also as expected).

```
root [0] gDebug=1
(int)1
root [1] TFile f("hsimple.root")
TKey Reading 307bytes at address 459104
root [2] TFile *f1 = new TFile("noot.root")
TKey Reading 47bytes at address 148
root [3] gROOT.Reset()
TFile dtor called for hsimple.root
TDirectory dtor called for hsimple.root
```

Debugging Macros

A powerful feature of CINT is the ability to debug interpreted functions by means of setting breakpoints and being able to single step through the code and print variable values on the way. Assume we have `macro4.C` still loaded, we can then do:

```
root [1] .b TChild::Print
Break point set to line 26 macro4.C
root [2] a.Print()

26   TChild::Print() const
27   {
28       cout << "This is TChild::Print()" << endl;
FILE:macro4.C LINE:28 cint> .s

311  operator<<(ostream& ostr,G__CINT_ENDL& i)
{return(endl(ostr));
FILE:iostream.h LINE:311 cint> .s
}
This is TChild::Print()

29     MyClass::Print();
FILE:macro4.C LINE:29 cint> .s

16   MyClass::Print() const
17   {
18       cout << "fX = " << fX << ", fY = " << fY << endl;
FILE:macro4.C LINE:18 cint> .p fX
(float)1.0000000000000e+01
FILE:macro4.C LINE:18 cint> .s

311  operator<<(ostream& ostr,G__CINT_ENDL& i)
{return(endl(ostr));
FILE:iostream.h LINE:311 cint> .s
}
fX = 10, fY = 12

19   }

30   }

2   }
root [3] .q
```

ROOT/CINT Extensions to C++

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

```
root [1] f = new TFile("hsimple.root")
(class TFile*)0x4045e690
root [2] f.ls()
TFile**          hsimple.root
TFile*           hsimple.root
  KEY: TH1F       hpx;1    This is the px distribution
  KEY: TH2F       hpxpy;1  py ps px
  KEY: THProfile  hprof;1  Profile of pz versus px
  KEY: TNTuple    ntuple;1    Demo ntuple
root [3] hpx.Draw()
NULL
Warning in <MakeDefCanvas>: creating a default canvas
with name c1
root [4] .q
```

The **first** command shows the first extension; the declaration of `f` may be omitted when "new" is used. CINT will correctly create `f` as pointer to object of class `TFile`.

The **second** extension is shown in the second command. Although `f` is a pointer to `TFile` we don't have to use the pointer dereferencing syntax `"->"` but can use the simple `"."` notation.

The **third** extension is more important. In case CINT can not find an object being referenced it will ask ROOT to search for an object with an identical name in the search path defined by `TROOT::FindObject()`. If ROOT finds the object it returns CINT a pointer to this object and a pointer to its class definition and CINT will execute the requested member function. This shortcut is quite natural for an interactive system and saves a lot of typing, e.g.:

```
root [4] TH1 *hpx = (TH1*)gROOT.FindObject("hpx")
root [5] hpx.Draw()
```

Of course when writing large macros, it is best to stay away from these shortcuts since otherwise you will later have problems compiling your macros using a real C++ compiler.

Intermezzo: Interpreting and Compiling a Macro

With some simple #ifdef's one can instrument a macro so it can be either interpreted by root (the executable) or compiled and linked with ROOT (the libraries):

```
#ifndef __CINT__

#include <stdio.h>
#include "Root.h"
#include "Class.h"
#include "Method.h"
#include "ClassTable.h"
#include "Collection.h"

#endif

void listmemfun(char *cls = 0)
{
    ...
}

#ifdef __CINT__

// Initialize the ROOT framework
TROOT api("TestApi", "Test CINT API");

int main()
{
    listmemfun("TObject");
    listmemfun("TClassTable");

    return 0;
}

#endif
```

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**C**lasses()
- gROOT->GetListOf**T**ypes()
- gROOT->GetListOf**G**lobals()
- gROOT->GetListOf**G**lobal**F**unctions()
- gROOT->GetListOf**F**iles()
- gROOT->GetListOf**S**ockets()
- gROOT->GetListOf**C**anvases()
- gROOT->GetListOf**S**tyles()
- gROOT->GetListOf**C**olors()
- gROOT->GetListOf**F**unctions()
- gROOT->GetListOf**G**eometries()
- gROOT->GetListOf**B**rowsers()

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 responsibility 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.

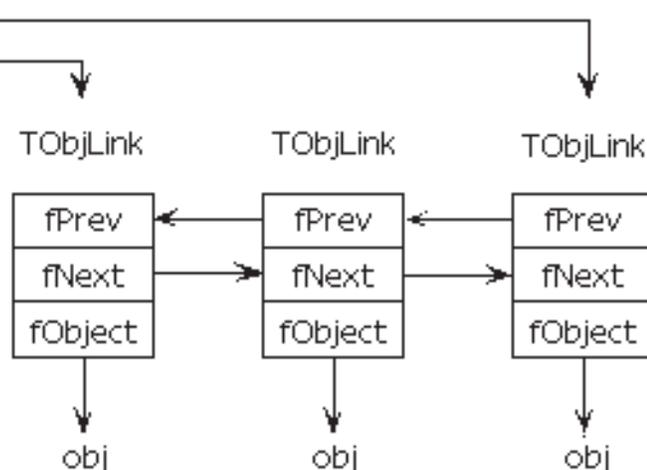
Hash() needs to be implemented by all objects that need to be stored in a hashtable (`THashTable`, `THashList` and `TMap`). By default `TObject::Hash()` returns the address of the object. It is essential to choose a good hash function.

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;
    TObj *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);  
}
```

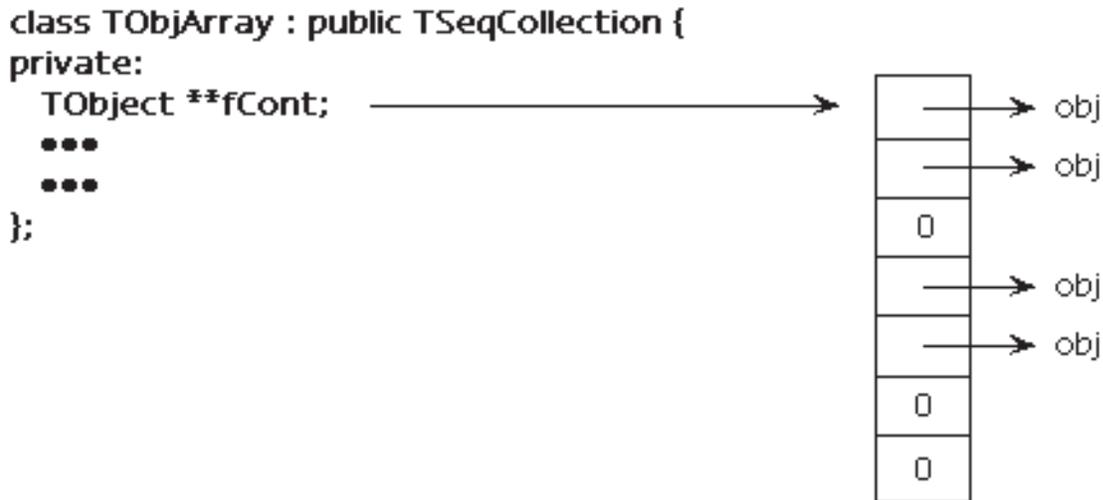
Method 1 uses internally method 2.

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:

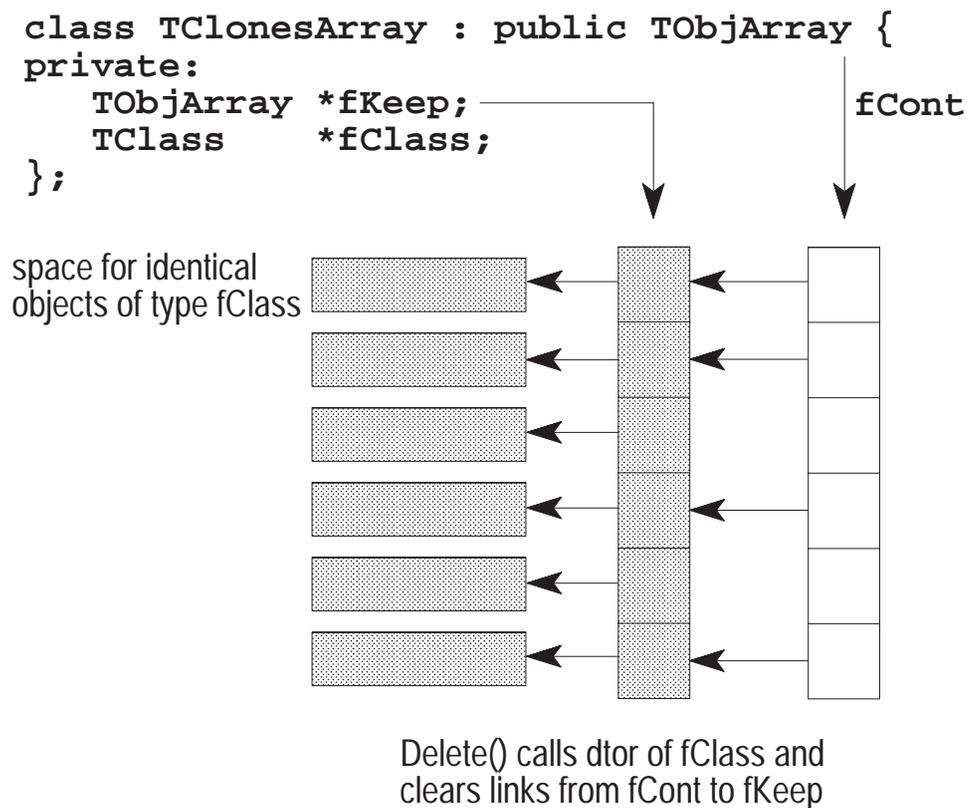
```
1 TIter
2 for (int i = 0; i < fArr.GetLast(); i++)
    if ((track = (TTrack*)fArr[i])) // or fArr.At(i)
        track->Draw();
```

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.



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 ($O(100000) \times O(10000)$ 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 $70 \mu\text{s}$, $O(10^9)$ 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 `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). 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 **TH2p** classes derive from the corresponding **TH1p** classes and also from the auxiliary class **TH2**.

The **TH3p** classes derive from the corresponding **TH1p** 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/hldraw.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 \mathbf{dp} &` `operator=(TH \mathbf{dp} &)`
- `TH \mathbf{dp}` `operator+(TH \mathbf{dp} &, TH \mathbf{dp} &)`
- `TH \mathbf{dp}` `operator-(TH \mathbf{dp} &, TH \mathbf{dp} &)`
- `TH \mathbf{dp}` `operator*(TH \mathbf{dp} &, TH \mathbf{dp} &)`
- `TH \mathbf{dp}` `operator/(TH \mathbf{dp} &, TH \mathbf{dp} &)`
- `TH \mathbf{dp}` `operator*(TH \mathbf{dp} &, Float_t)`

Where \mathbf{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 `THdF`'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 rewrite 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 `TH1p` 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:
$$f(x) = p_0 \cdot \exp(-0.5 \cdot ((x-p_1)/p_2)^2)$$
- "expo" An exponential with 2 parameters:
$$f(x) = \exp(p_0 + p_1 \cdot x)$$
- "polN" A polynomial of degree N :
$$f(x) = p_0 + p_1 \cdot x + p_2 \cdot x^2 + \dots$$

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");

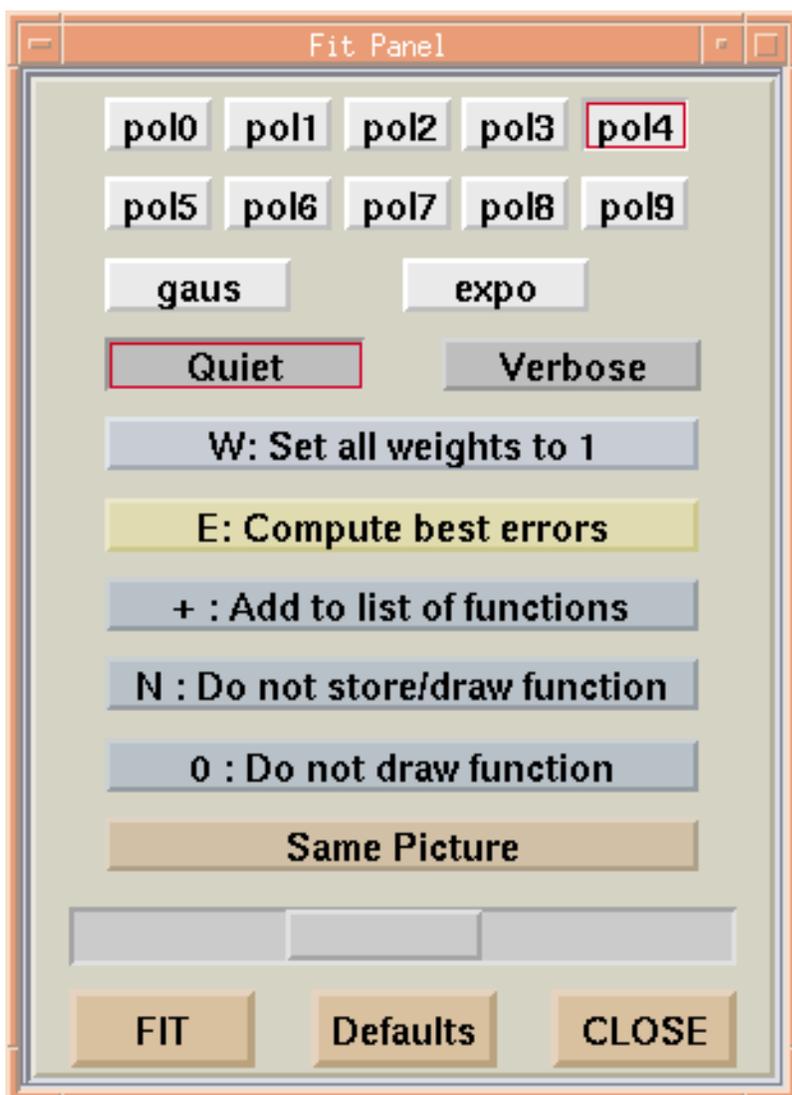
    TH1F *hpx = (TH1F*)f->Get("hpx");

    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`. Accessible via the histogram context menu.

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 an 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 let's 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 sub-directories 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 (ala 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 hpypy;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, const 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