# 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.**

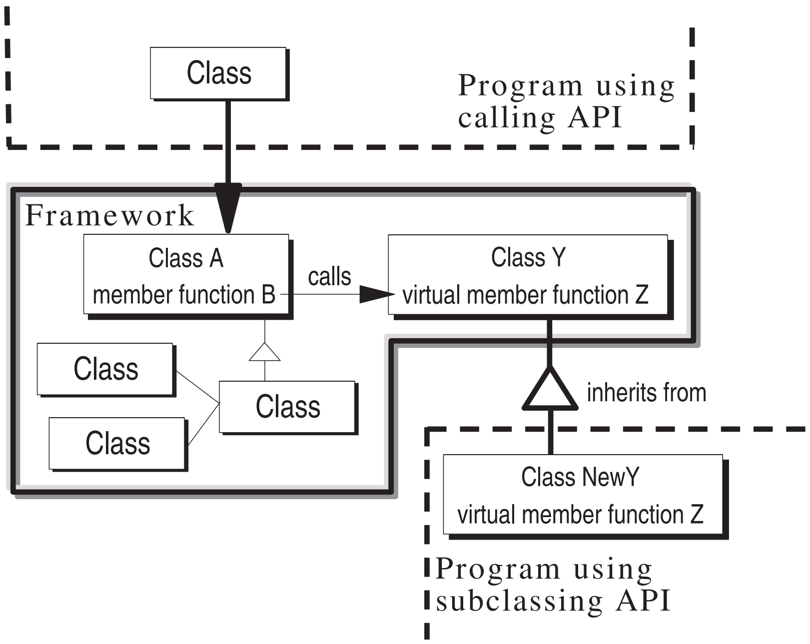



*Figure 2  Calling API versus subclassing API*

# Advantages of Frameworks

**The benefits of frameworks can be summarized as follows:**

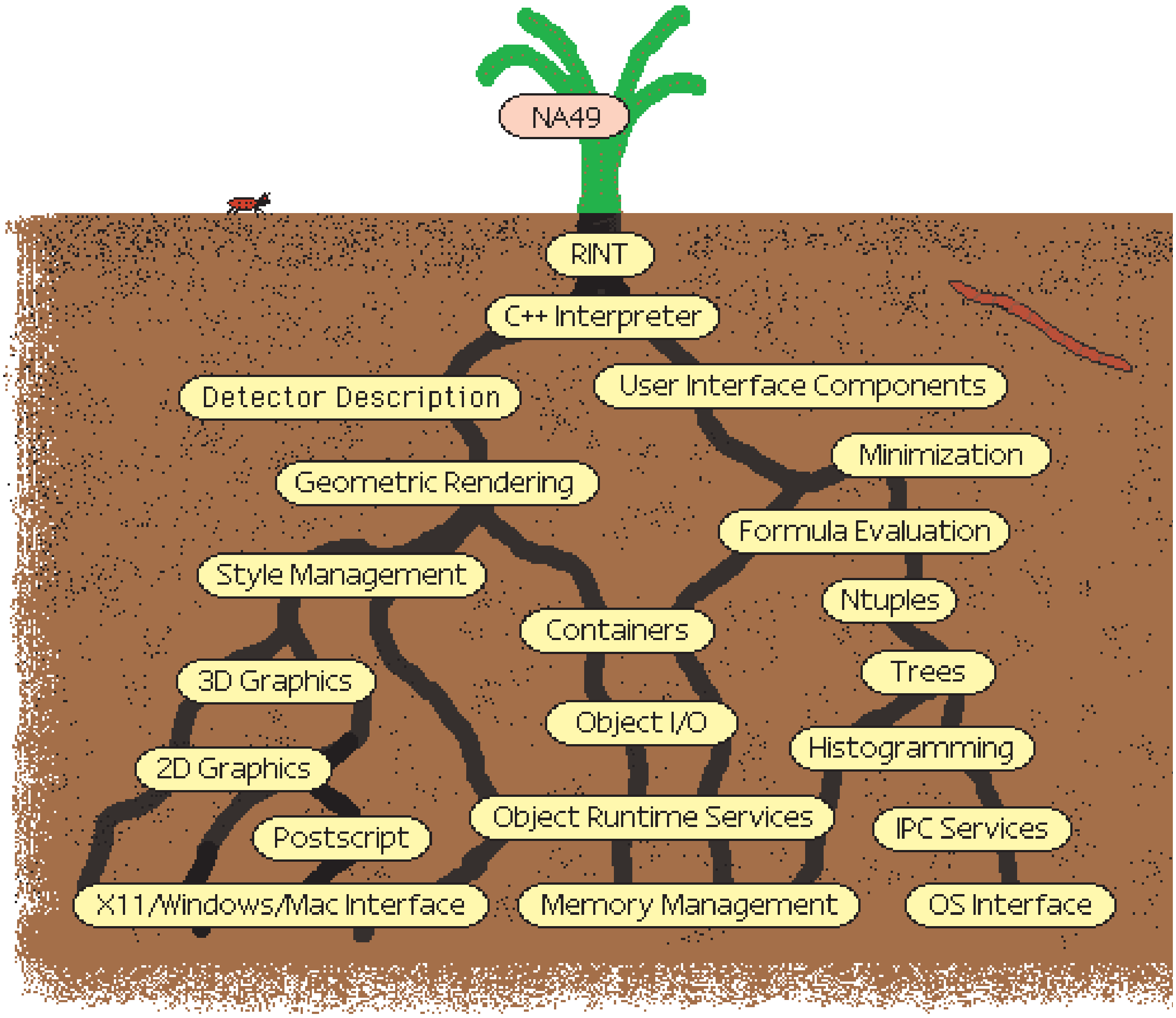
- **Less code to write**. Much of the program's design and structure, as well as its code, already exist in the framework.

- **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]
```

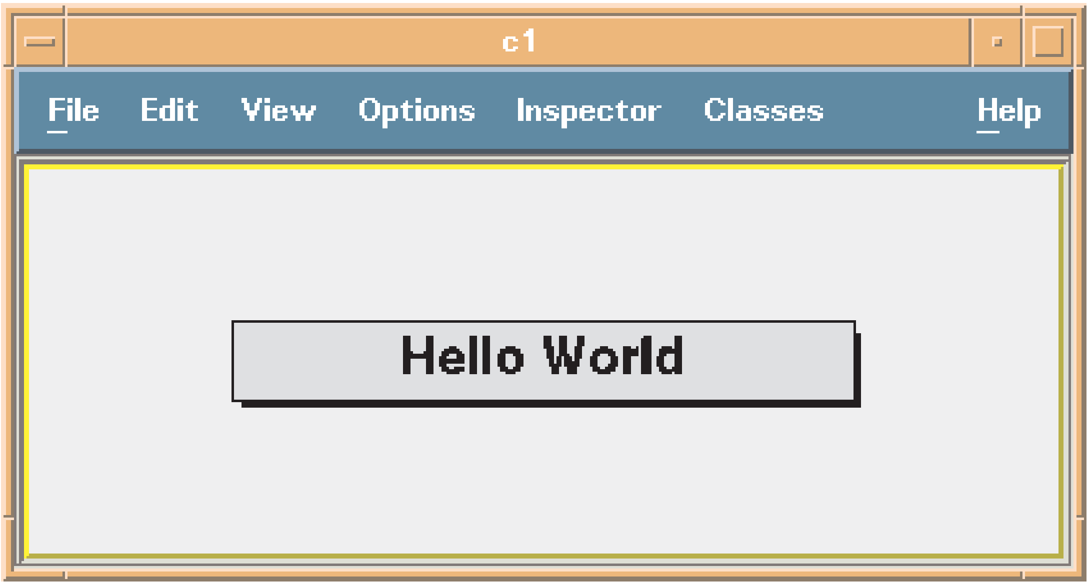
# Running the Interactive ROOT Program

**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()`. I must exists 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 the 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()`, `GetFirst()`, `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

```
(hproot) [199] root
  *******************************************
  *                                         *
  *        W E L C O M E  to  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 *ll = new TLine
root [3] TPolyLine *ll = new TPolyLine
Error: ll 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)1.000000000000e+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)
{
   ...
   ...
}

#ifndef __CINT__

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

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

   return 0;
}

#endif
```