A Short Course on ROOT Day 1

Subir Sarkar
SINP, Kolkata

VIIIth SERC School on EHEP, VECC, Kolkata
June 20 – July 10, 2011

Course Schedule – Day 1

- ROOT at a glance
- CINT interactive sessions
  - Interpreter
  - **A**utomatic **C**ompiler of **Li**braries for **C**INT (ACLiC)
- ROOT Libraries
  - TObject, ROOT inheritance tree
  - TROOT
- Adding external classes to the system
- ROOT Globals
- Collection classes

ROOT at a Glance

- ROOT is an OO C++ framework developed for large scale data handling that provides
  - an efficient data storage and access system designed to support structured datasets of PetaByte scale
  - a C++ interpreter
  - histogramming and fitting
  - advanced statistical analysis algorithms (multi dimensional histograms, fitting, minimization, cluster finding etc.)
  - collection classes
  - scientific visualization tools with 2D and 3D graphics
  - extensive Run-Time Type Information (RTTI)
  - graphical application development framework
  - geometry modeller
  - PROOF parallel query engine

ROOT at a Glance

- The user interacts with ROOT via
  - a graphical user interface
  - the command line
  - C++ scripts
  - compiled programs
- Thanks to the embedded CINT C++ interpreter, both command line and scripting language is C++
- Large scripts should be compiled and dynamically loaded
- The ROOT library can be accessed seamlessly from Python/Ruby as well

Root at a Glance

- User classes
  - user can define new classes interactively
  - use either calling API or sub-classing API
  - these classes can inherit from ROOT classes
- Dynamic linking
  - interpreted code can call compiled code
  - compiled code can call interpreted code
  - macros can be dynamically compiled & linked

normal mode of operation

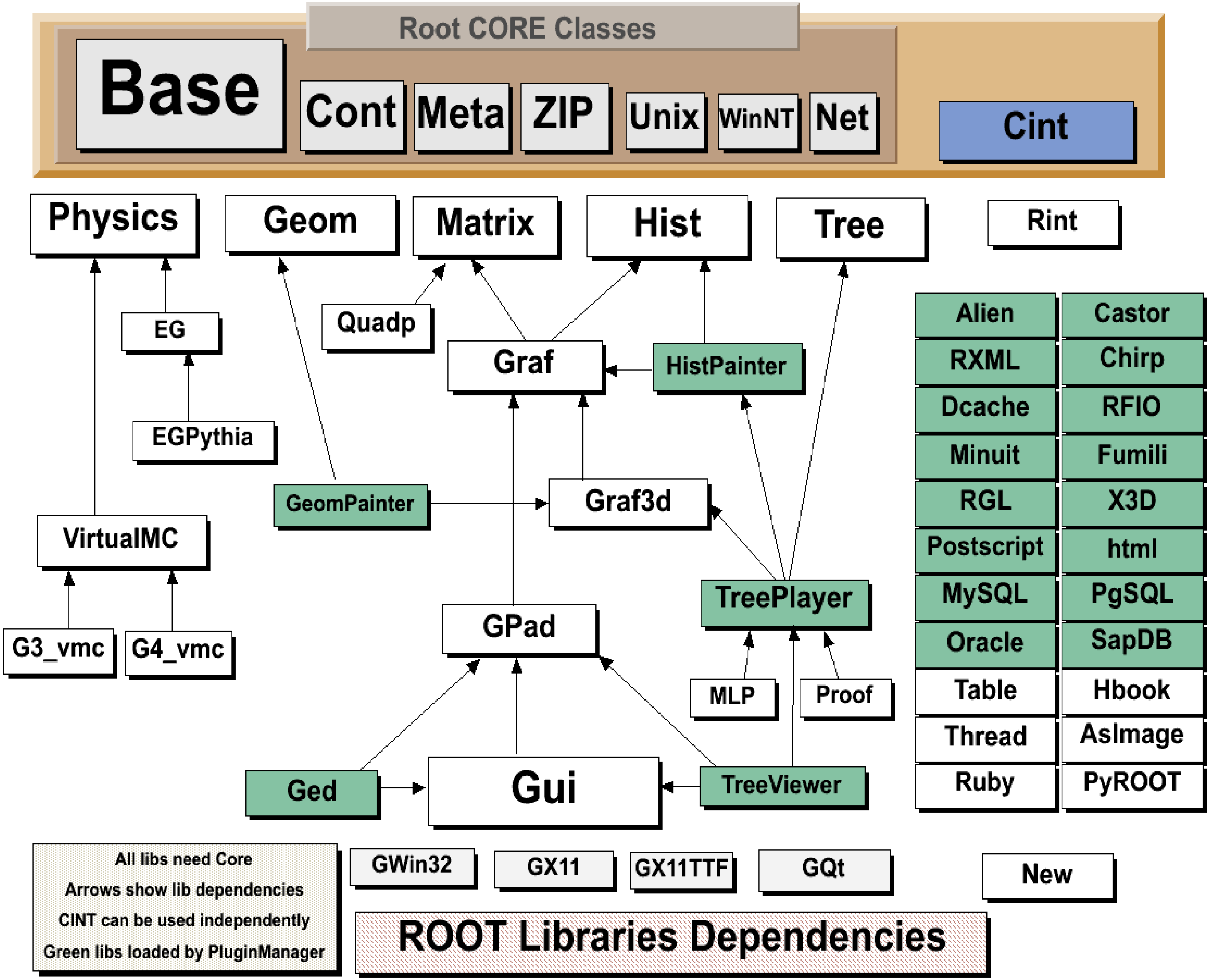
extend an application dynamically

script compilation
root > .x file.C++

ROOT Library Structure

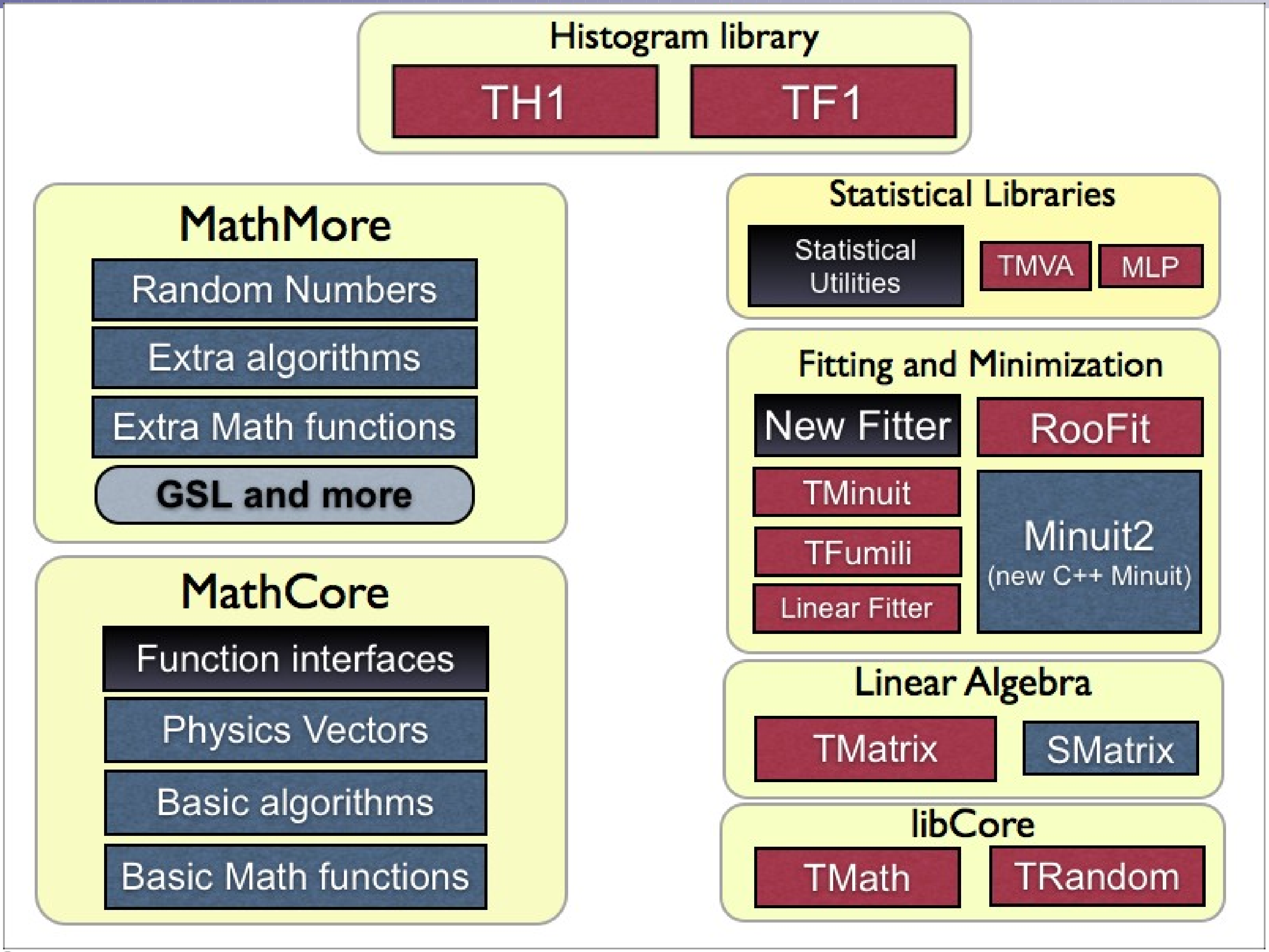
- ROOT libraries are arranged in a layered structure
- The Core classes are always required (support for Run Time Type Information, basic I/O & interpreter)
- The optional libraries (you load only what you use)
  - separation between data objects and the high level classes acting on these objects.
  - example, a batch job uses only the histogram library, no need to link histogram painter library
- Why shared libraries?
  - reduce the application link time
  - reduce the application size
  - can be used with other class libraries
  - usually loaded via the plug-in manager

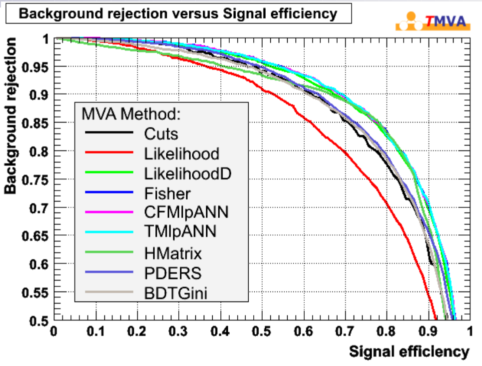
The ROOT Libraries

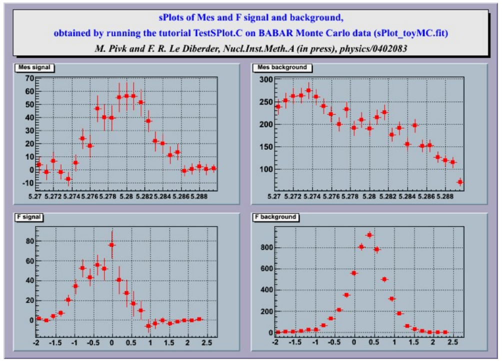


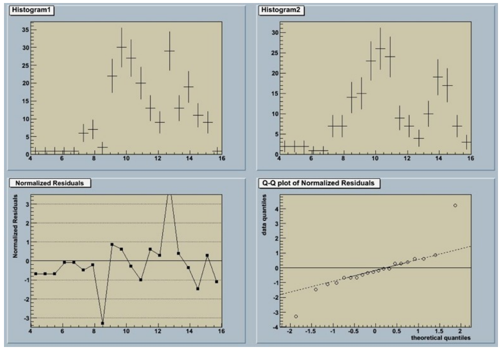
- Over 1500 classes
- ~ 1.5M lines of code
- CORE (8 Mbytes)
- CINT (2 Mbytes)
- Green libraries linked on demand via plug-in manager (only a subset shown)
- > 100 shared libs

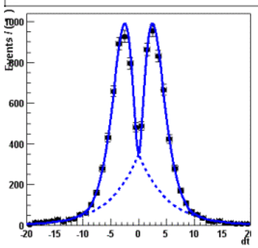
Math libraries

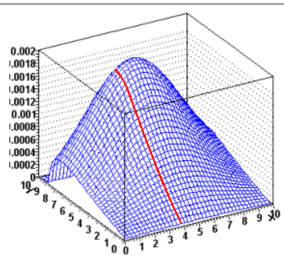


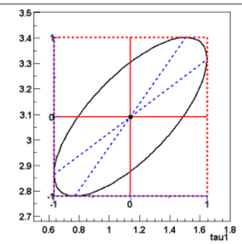








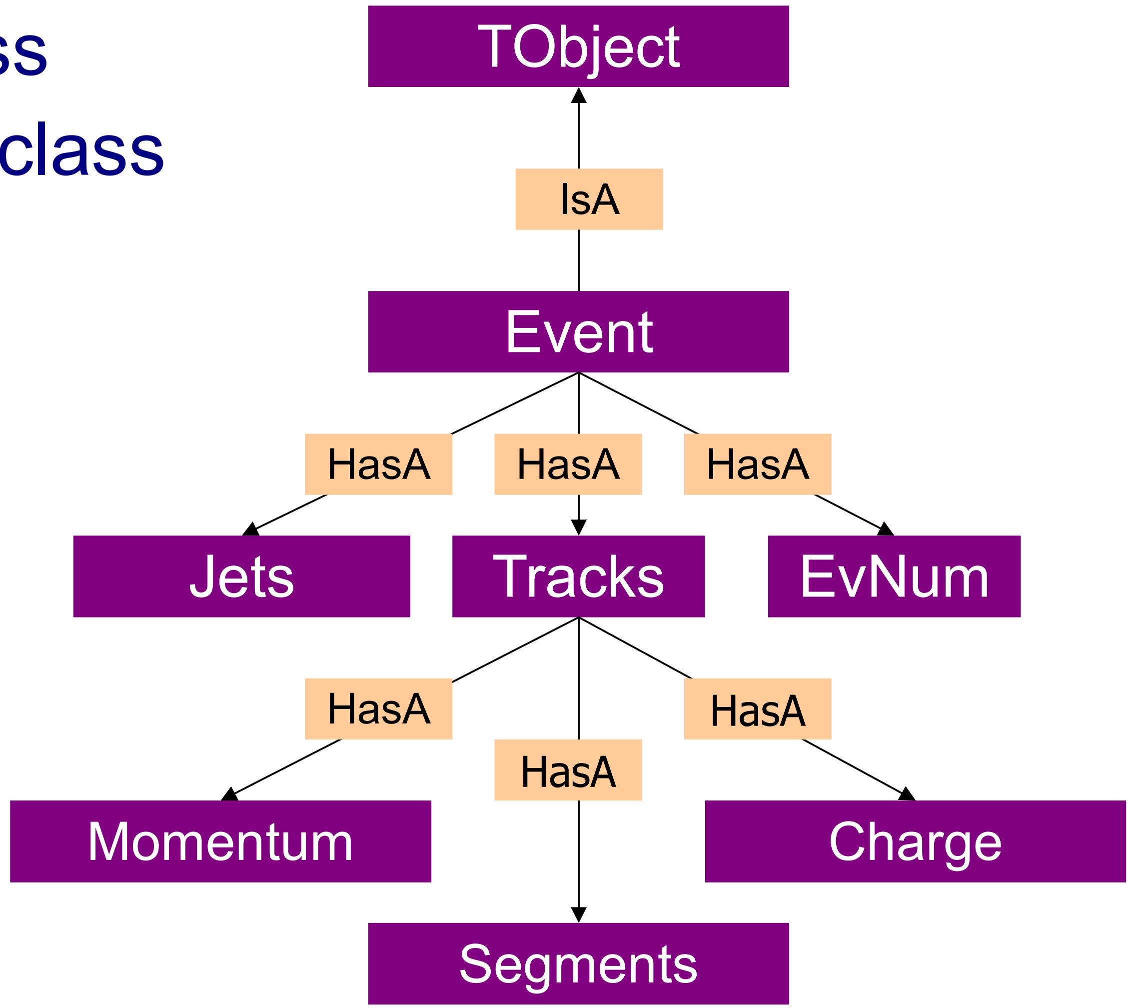




Object Orientation - Recap

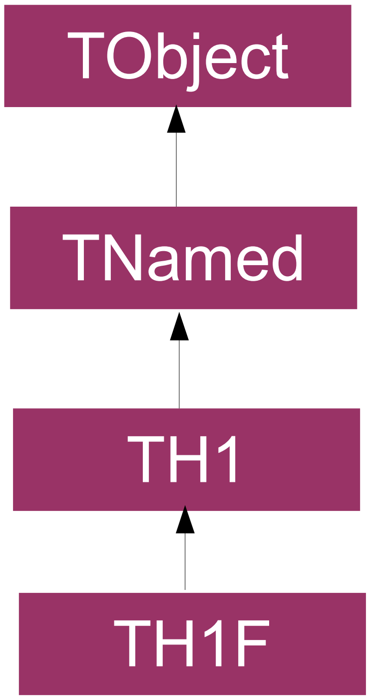
- Class: blueprint
- Object: instance of a class
- Methods: functions for a class

  - Members: a “has a” relationship to the class.
  - Inheritance: an “is a” relationship to the class.



TObject – The Base Class

- TObject provides default behavior and protocol for almost all objects in the ROOT system
  - object I/O (Read(), Write())
  - error Handling (Warning(), Error(), Fatal())
  - sorting (Compare(), IsEqual())
  - inspection(Dump(), Inspect())
  - drawing, printing
  - bit handling (SetBit(), TestBit())
  - meta information (IsA(), InheritsFrom())



- An object of any class that inherits from TObject can be made persistent (object I/O)

TROOT

- the TROOT object is the main entry point to the system
  - created as soon as the Core library gets loaded
  - initializes the rest of the ROOT system
  - a singleton, accessible via the global pointer gROOT
    - an omnipotent global, handle with care
  - provides many global services
    - `gROOT->GetListOfFiles()`
    - `gROOT->GetListOfCanvases()`
  - via gROOT you can find basically every object created by the system,

```
TH1F *hpx = (TH1F*) gROOT->FindObject("hpx") // C-style
```

```
TH1F *hpx = dynamic_cast<TH1F*>(gROOT->FindObject("hpx")) // C++ style
```

CINT in ROOT

- CINT is used in ROOT
  - as command line interpreter
  - as script interpreter
  - to generate class dictionaries
  - to generate function/method calling stubs
  - signals/slots with the GUI
- The command line, scripting and programming language become the same
- Large scripts can (and should) be compiled for optimal performance
  - a little care is required

Compiled vs Interpreted

- Why compile?
  - faster execution, CINT has (many) limitations…
  - compilation error diagnosis much better
  - code validation
- But then, why interpret?
  - faster Edit → Run → Check result → Edit cycles ("rapid prototyping").
  - scripting is sometimes just easier, specially for simple tasks
    - compilation overhead may be discouraging
  - ACLiC is even platform independent!

ROOT session - setup

```
export ROOTSYS=/opt/root
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
export PATH=$ROOTSYS/bin:$PATH
```

$ source /opt/root/bin/thisroot.(c)sh

.bashrc

Environment

- `$PWD/.rootrc`
- `$HOME/.rootrc`
- `$ROOTSYS/etc/system.rootrc`

```
root [] gEnv->Print()
```

Options are merged with the above order of precedence

```
Root.MemStat:            1   gObjectTable->Print()
Root.ObjectStat:         1
Browser.Name:            TRootBrowser
```

Starting a ROOT Session

```
[sarkar@localhost ~]$ root -h
Usage: root [-l] [-b] [-n] [-q] [dir] [[file:]data.root] [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
  -l : do not show splash screen
  -x : exit on exception
 dir : if dir is a valid directory cd to it before executing

  -?       : print usage
  -h       : print usage
  --help   : print usage
  -config  : print ./configure options
  -memstat : run with memory usage monitoring
```

- To run function mycode() in file mycode.C

```
root [] .x mycode.C
```

- Equivalent: load file and run function

```
root [] .L mycode.C
root [] mycode()
```

ROOT session - I

Calculator

“;” prevents printing the return code

```
root [0] gROOT->GetVersion()
(const char* 0xabf498)"5.28/00"
root [1] 344+76.8
(const double)4.20800000000000010e+002
root [2] float x=89.7;
root [3] float y=567.8;
root [4] x+sqrt(y)
(double)1.13528550991510710e+002
root [5] float z = x+2*sqrt(y/6);
root [6] z
(float)1.09155929565429690e+002
root [7] .q
```

Command history

See file $HOME/.root_hist

```
root [0] try up and down arrows
```

ROOT session - II

root

```
root [0] .x session2.C
for N=100000, sum= 45908.6
root [1] sum
(double)4.59085828512453370e+004
Root [2] r.Rndm()
(Double_t)8.29029321670533560e-001
root [3] .q
```

unnamed macro
executes in global scope

does not accept arguments

session2.C

```
{
   int N = 100000;
   TRandom r;
   double sum = 0;
   for (int i=0;i<N;i++) {
      sum += sin(r.Rndm());
   }
   printf("for N=%d, sum= %g\n",N,sum);
}
```

ROOT Session - III

root

```
root [0] .x session3.C
for N=100000, sum= 45908.6
root [1] sum
Error: Symbol sum is not defined in current scope
*** Interpreter error recovered ***
Root [2] .x session3.C(1000)
for N=1000, sum= 460.311
root [3] .qx
```

Named macro
Normal C++ scope
rules

session3.C

```
void session3 (int N=100000) {
   TRandom r;
   double sum = 0;
   for (int i=0;i<N;i++) {
      sum += sin(r.Rndm());
   }
   printf("for N=%d, sum=%g\n",N,sum);
}
```

Compiling Code: ACLiC

Automatic Compiler of Libraries for CINT

- Load code as shared lib, much faster

```
.x mymacro.C+(42)
```

- Uses the system's compiler, takes seconds
- Subsequent .x mymacro.C+(42) check for changes, only rebuild if needed
- Exactly as fast as Makefile based standalone binary!
- CINT knows types, functions in the file, e.g. call

```
mymacro(43)
```

ROOT session – IV (ACLiC)

Automatic Compiler of Libraries for CINT

```
root [0] gROOT->Time();
root [1] .x session4.C(10000000)
for N=10000000, sum= 4.59765e+006
Real time 0:00:06, CP time 6.890
root [2] .x session4.C+(10000000)
for N=10000000, sum= 4.59765e+006
Real time 0:00:09, CP time 1.062

root [3] session4(10000000)
for N=10000000, sum= 4.59765e+006
Real time 0:00:01, CP time 1.052
root [4] .q
```

CINT knows all functions in session4_C.so/.dll

session4.C

```
#ifndef __CINT__
#include "TRandom.h"
#endif
void session4 (int N) {
   TRandom r;
   double sum = 0;
   for (int i=0;i<N;i++) {
      sum += sin(r.Rndm());
   }
   printf("for N=%d, sum= %g\n",N,sum);
}
```

File session4.C
Automatically compiled and linked by the native compiler.
Must be C++ compliant

ACLiC Options

```
root [] .L MyScript.C+
root [] .L MyScript.C++ // Force recompilation
root [] .L MyScript.C+(+)g
root [] .L MyScript.C+(+)0

root [] gSystem->SetAclicMode(TSystem::kDebug);
root [] gSystem->SetAclicMode(TSystem::kOpt);

root [] gROOT->ProcessLine(“MyScript.C+”);
root [] gROOT->LoadMacro(“MyScript.C+|++[g|0]”);
root [] gROOT->Macro(“MyScript.C+|++[g|0]”);

root [] gSystem->CompileMacro(“MyScript.cxx”,”f”);
                                      k|f|g|O|c
root [] gSystem->Load(“lib/mylib”); //libmylib.so
root [] gSystem->SetIncludePath(“-I./app/include”);
```

ROOT session - V
macro with more than one function

```
root [0] .x session5.C >session5.log
root [1] .q
```

```
root [0] .L session5.C
root [1] session5(100); >session5.log
root [2] session5b(3)
sum(0) = 0
sum(1) = 1
sum(2) = 3
root [3] .q
```

session5.C

```
void session5(int N=100) {
   session5a(N);
   session5b(N);
   gROOT->ProcessLine(".x session4.C+(1000)");
}
void session5a(int N) {
   for (int i=0;i<N;i++) {
      printf("sqrt(%d) = %g\n",i,sqrt(i));
   }
}
void session5b(int N) {
   double sum = 0;
   for (int i=0;i<N;i++) {
      sum += i;
      printf("sum(%d) = %g\n",i,sum);
   }
}
```

.x session5.C
executes the function
session5 in session5.C

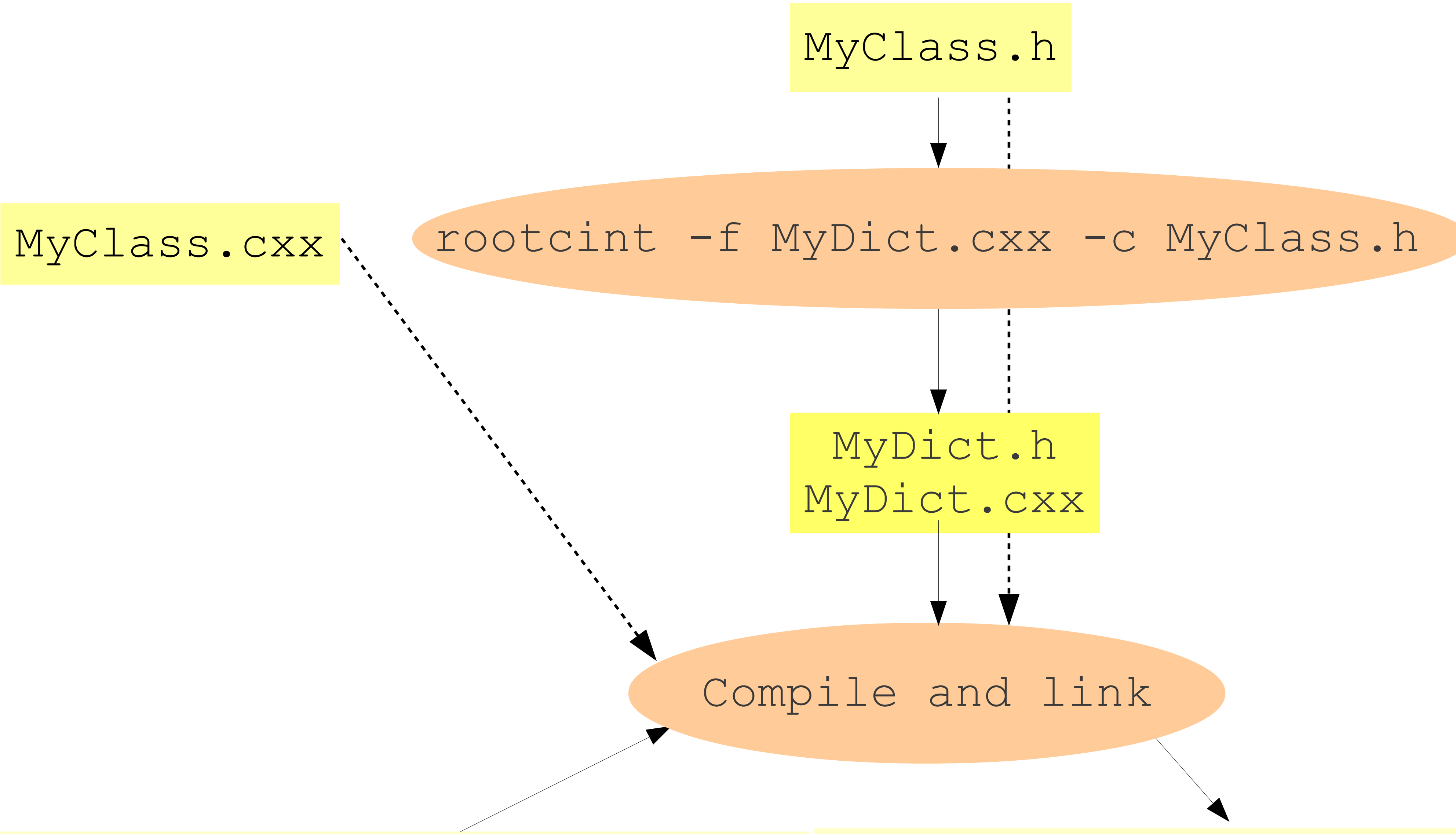
use gROOT->ProcessLine
to execute a macro from a
macro or from compiled
code

Integrating User Classes to the system

- develop MyClass
- generate a dictionary for MyClass
  - generation of Streamer() methods (I/O)
  - generation of ShowMember() methods (run-time object inspection)
  - provision for extended RTTI
- compile and link – create the shared library
- execute MyClass methods within the interpreter
- save and restore MyClass objects

```
#include <iostream>
using namespace std;
class MyClass {  // For simplicity we do not allow any extension
private:
  float fX;
  float fY;
public:
  MyClass() {fX = fY = -1;}
  void SetX(float X) {fX = X;}
  void SetY(float Y) {fY = Y;}
  void Print() const {cout<< "fX = "<< fX << ", fY = " << fY <<  endl;
};
```

Integrating User Classes




```
g++ -shared -fPIC `root-config \
--cflags` -o libMyClass.so \
MyClass.cxx MyDict.cxx
```

```
root [0] gSystem->Load("libMyClass")
root [1] MyClass a
root [2] a.Print()
```

Linking Class to ROOT RTTI

- To link a class to the ROOT RTTI system two macros need to be added to MyClass
  - ClassDef(*class name*, *version id*)

```
// MyClass.h
class MyClass {
public:
    . . .
   ClassDef(MyClass,1)   // analyse my data
};
```

  - ClassImp(*class name*)

```
// MyClass.cxx
#include "MyClass.h"
ClassImp(MyClass)
```

Run Time Type Information (RTTI)

- add: `#include <Rtypes.h>` to MyClass.h
- repeat the same steps as previously

```
> rootcint -f MyDict.cxx -c MyClass.h
```

```
> g++ -shared -fPIC `root-config --cflags` -o libMyClass.so MyClass.cxx MyDict.cxx
```

```
> root
root [0] gSystem->Load("libMyClass")
```

```
root [1] MyClass a
root [2] a.Print()
root [3] a.P<TAB>    works now
```

Integration into ROOT

- To behave as a fully ROOT supported class, with full I/O capabilities, just add derivation from TObject to MyClass

```
// MyClass.h
class MyClass : public TObject {
public:
   . . .
   ClassDef(MyClass,1)  // analyse my data
};
```

Integration into ROOT

- Create object of MyClass and Dump() its run time contents

```
root [1] MyClass a; // on stack
root [2] a.Dump();
root [3] …
```

```
root [1] MyClass a;
root [2] TFile f(“test.root”, “recreate”);// stack
root [3] a.Write(“a1”);
root [4] f.Close();
```

```
root [1] TFile f(“test.root”);
root [2] MyClass *a = \
          dynamic_cast<MyClass*>(f.Get(“a1”));
root [3] f.Close();
root [4] a->Dump();
```

CINT Extensions to C++

```
root[] f = new TFile("test.root");
root[] f.ls();
```

```
root[] TFile* f = new TFile("test.root");
root[] f->ls();
```

```
root[] hpx.Draw();
```

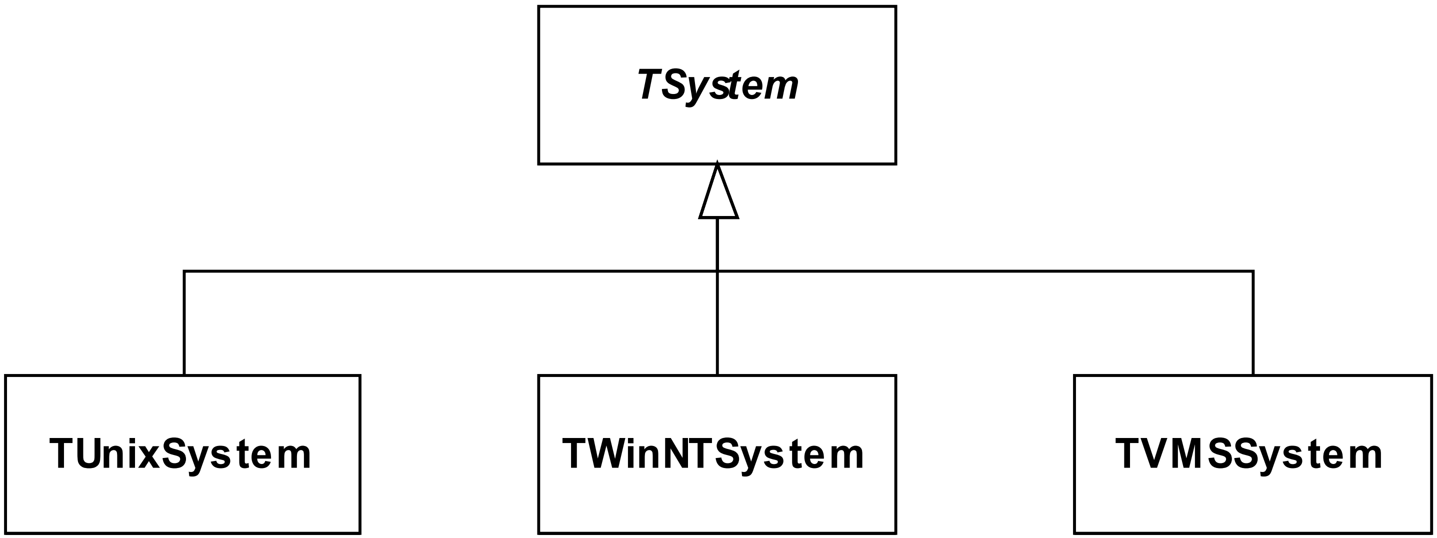
```
root[] THF1* hpx =
          dynamic_cast<TH1F*>(f->Get("hpx"));
root[] hpx->Draw();
```

```
root[] a=2 //declaration,semicolon not needed
Warning: Automatic variable a is allocated
(tmpfile):1: (const int)2
```

The above extensions do not work when complied

Operating System Interface

- The underlying OS is abstracted via the TSystem abstract base class
- Accessible via the gSystem singleton
- It allows all ROOT and user code to be OS independent



TSystem Services

- Environment variable manipulation
  - getenv, putenv, unsetenv
- System logging
  - syslog interface
- Dynamic loading
  - load, unload, find symbol, …
- File system access
  - file creation and manipulation
  - directory creation, reading, manipulation
- Please check TSystem carefully for the right methods; keep your code portable