Scripting in the ROOT analysis framework

Tomasz Bold
tomasz.bold@fis.agh.edu.pl
D11 – room 107

Today

- The ROOT environment
  - general overview (even experts may hear something new)
  - the CINT/CLING interpreters
- from the script to the executable
  - anonymous and named scripts
  - on the fly compiled code
  - using code from the compiled library
  - executable code

Pre ROOT analysis frameworks

- A bit of history ☺
  The particle/nuclear physics community used to challenges: data volumes/advanced statistical analysis/publication quality. In the times of the “H.M. King Fortran 77th” there was not so many (if any) systems for analysing big data sets: data format → analysis → presentation.
- The PAW (Physics Analysis Workstation) was the answer
  - equipped with data storage API for histograms and n-tuples
  - fully interfaced to CERNLIB for statistical analysis
  - state of art fitting
  - very good quality/and rich data visualisation (1/2/3-D plots)

ROOT

- It has become fashionable to use C++ in early 1990 in HEP
  - there were serious competitors i.e. Fortran 90, pure C
  - there were also c++ competitors: HEP++

- PAW team decided to try c++
  - first a lot of interfacing between CERNLIB(f77) and c were involved
  - but then gradually more and more codes were (re)written in c++
    - sometimes by students
    - sometimes just by looking at the Fortran and translating to c++ ☺
  - Anyways project was well received and accepted
    - also because ETH was very fast

Written mainly at CERN by group led by Rene Brun.
Now big group of contributors also in other HEP and other domains

Key ROOT ingredients

- The ROOT file formats
  - sufficiently versatile to store:
    - histograms (1/2/3-D, with associated graphical properties)
    - n-tuples (trees) – for novices this is like sequence of DB records important optimizations were put in place in the form of ROW-wise and COLUMN-wise ntuples
    - adding user defined data (essential functionality) very easy
      - it involves going over the class members and marking which members are not needed + some straightforward keywords to be added here and there
      - NOTE. This was c++ persistency system available years before anything like Boost serialization appeared on the market, and even now it is one of the fastest.

- The **Cling** interpreter!
  - In the older ROOT5 CINT → supporting a flavour of c++.

ROOT components

- The ROOT is an executable/interpreter + collection of libraries
  the libraries can be found at $ROOTSYS/lib/lib*.so
  - Core – the essentials
  - Tree – the objects storage
  - Hist – histogramming
  - MathCore – basic mathematics (TMath class)
    - MathMore - ☺ more math. stuff
  - Minuit – minimalization (fitting)
  - Proof* - Parallel events processing
  - Qt* - Qt library interfacing
  - GPad/TG* - GUI
  - RooFit/Stat/ - fitting
  - TMVA – multivariate analysis
  - + collection of low level IO libraries (RFIO,dcache, mysql, ...)

- Every element in fact deserves one lecture in order to describe it well

Where to find info

- root.cern.ch – there you find user manual and description of the system class by class
- lot's of examples and useful & responsive users forum

The Cling interpreter

- The success of ROOT is 80% (in my view) is due to its build in interpreter
- It offers sufficient subset of c++ functionality
- build in reflexion allows discovery of objects API in the prompt

The Cling interpreter

To setup the ROOT, you need to know the directory in which it is installed (usually /usr/local/root).
Then it is enough to **source /usr/local/root/bin/thisroot.sh**

$ root

root [0] cout << "Hello world\n”

Hello world

In lab nothing is needed, just type root in the terminal

- The root invocation options :
  - -l – no initial banner * -b – batch mode, no graphics * -q – quit after processing the script
  - **root file(s)** – files will be opened (read only mode)
  - **script file** – the file with commands interpreted

- The interpreter accepts c++ language constructs and the .(dot) commands
  - .? – help
  - .ls – list file content
  - .T – tracking on .....

Type what’s in **red**

Accessing content of the ROOT files

- after calling: **root a.root b.root c.root**

- the interpreter will create variables **_file0, _file1 and _file2** associated with corresponding files a/b/c.root
  - you can change to any of it by **_file1.cd() (or _file0->cd())**
  - the file may contain directories: change to them by **dir.cd() (dir->cd())**
  - there is always global variable **gDirectory** pointing to the current directory

- The objects in the file can be accessed by name:
  - i.e. if there is histogram named **hGauss**
  - **hGauss->Draw()** → to draw it
  - **hGauss->GetEntries()** → to see how many entries were there etc.

- Once you use the name **hGauss** the in memory copy is made of that histogram and it can be changed

Accessing content of the ROOT files

- Smart things are going one when you type:
  root> **hGauss->Draw()**

- First of all the variable is created
  - this is c++ so it's type has to be deferred
  - it's methods have to be figured out
  - appropriate bytes copied from disk to memory

- All this is done by the **reflexion** mechanisms
  - each stored object has to be made known to ROOT
  - this is done by generating it's dictionaries
  - This is described in more details in the users manual (together with examples & recipe)

Adding content to the ROOT files

- The file in append/write mode can be opened in the session in several ways. Most portable/convenient is this:
  file = TFile::Open(”output.root”, ”RECREATE”) // or ”UPDATE”
  - It can be also used with **”OLD”** for reading and then it accepts various protocols i.e. **http, rfio, ...**

- whatever is created (i.e. histogram) gets context of the current file → it is enough then to do
  file->Write(); file->Close();
  to have things written.

ROOT scripts

- The actions (i.e. histograms drawing etc.) can be grouped in the scripts
  - often the ultimate goal is to compile the content of this script and obtain an executable program (but not always, i.e. drawing)
  - there is 5 levels (my personal view) between fully interactive to fully compiled
    1. anonymous scripts – micro projects
    2. named scripts – a bit bigger stuff
    3. on the fly compiled scripts – when speed matters
    4. externally compiled libraries
    5. standalone applications
  - The hybrid solutions are often used
    - i.e. driving script is the anonymous and data cruncher an external library

- We will go over the details of mix&match of all this

1

Anonymous script

- The script file name should match: *.C
- content starts from the **{** in the first line & first character of the file and ends with the **}**

```
{
  std::cout << „hello”;
}
```

- whatever worked in the root prompt will work also here
- Example of use:
  root –l a.root MakePlots.C
  or:
  $ **root –l a.root**
  root[o] .x MakePLots.C
- After the lines are evaluated all the variables (i.e. histograms) created are available for interactive inspection or further processing
  - i.e. sequence of scripts can be invoked

2 Named scripts

- When the task becomes too big for few lines of the script it needs to be split into the functions —> Named scripts become handy.
  - Usually have arguments,
  - Reusable

- The file named **xyz.C** **has to** contain function **xyz**
  - the execution starts from this function
  - additional functions can be defined in such a script
  - after the function is executed the prompt is given back
  - internal variables are not accessible, out-of-scope objects deletion is performed etc.

- the named script can be invoked in few ways
  - loaded and then executed
    .L xyz.C
    xyz()
  - loaded in and executed in one shot
    .x xyz.C
  - invoked from the command line
    root a.root xyz.C

if script needs to be loaded in another script:
gROOT->ProcessLine(".L blah1.C")

2

Named scripts
arguments and return types

- The **xyz** function can have arguments and/or return objects
  - the returned object can be of any type (make sure it is not getting out of scope)
- arguments also can be of any type
  - interesting trick can be made in invoking from the command line:
    root –q –l a.root 'xyz.C("output256.root", 256, 0.3)'
  - the function would need to be then defined as follows:
    void xyz (const char* output, int iterations, float step) {…
  - this readily solves common issue of code generalization

3 Compiled named scripts

on the fly compilations

- In case the script needs to be fast it can be compiled before the running using ACLIC (automatic compilation and linking)
  - It is enough to add + or ++ in all the commands
    .L xyz.C++
    .x xyz.C+("056.root", 56, 0.1)
    root –q –l –b a.root 'xyz.C+("018.root", 18, 0.8)'
  - **+** - compile if source changed
  - **++** - recompile always
  - **+g** – add debugging symbols (useful when script is crashing)
  - **+O** – optimized compilation

If compiled script needs to be loaded in another script:
gROOT->ProcessLine(".L xyz.C+")

- Once compiled, the code is placed in **xyz_C.so** and can be loaded with **.L blah1_c.so**

- For the compilation to be successful the named script must be valid c++
  - the compilation process assures better code quality

4

Libraries

- When the project grows the ACLIC compilation may be inconvenient
  - it is possible to add external libraries in fact via:
    gSystem->AddLinkedLibs (...)
    gSystem->AddIncludePath(...)
    ...
- It may be more convenient to make the library with the regular c++ compiler (i.e. the g++)
- The loading library is easy: **".L"**
- But how can we use components from that library in Cling?

4

Accessing components from the compiled libraries

- Each function, class which is intended for access in the interactive (not compiled) code needs description. → the dictionary.

- ROOT this days uses Reflex
  - it takes the source code (what is in the header in fact) and generates description
    → using this description the name **"xyz"** can be mapped to the "**xyz"** function
  - **rootcint header1.h header1.h [LinkDef.h] > dict.C**
  - The **dict.C** needs to be compiled into the library

- While generating the dictionary we can tell which objects are of interest for us
  - The directives file, c++ header file, usually called **LinkDef.h**

```
#ifdef __CINT__
#pragma link C++ function xyz;
#pragma link C++ class Indices-;
#pragma link C++ class MyStat+;
#endif
```

- Library made this way can be then loaded to ROOT interactive session and **xyz** called, objects of class **MyStat** created, saved in the files etc.

5 Standalone application

- We now departed from the script and are to make standalone program
- This is as usual as writing any standalone C++ program.
  - ROOT comes with the **root-config** which provides compiler options i.e. **root-config –libs –clags**
  - The reflexion mechanisms may seem useless ... hoever if you want to store classes in the files you need streamers and so playing with **rootcint** will be necessary.

Summary

- The ROOT CINT/CLING has nice learning curve:
  - Start from command line (with build in help)
  - Anonymous scripts
  - Named scripts (this code becomes testable)
  - Compiled named scripts (fairly good level of validation)
  - Externally compiled libraries
  - Standalone applications

- Testing of ROOT macros is usually not emphasized
  - In constant flux (change as the data analysis evolves)
  - Usually, each analyzer gets (at least) to the compiled script stage which provides decent level of code validation!
  - Bigger projects are usually tested at bigger level (integration tests)

Addendum: setting ROOT on taurus server:

- There are two version of the root installed. System one which is available just as any other program.
- There is the newest version also available:
- **source ~tbold/ROOT/bin/thisroot.sh**