Introduction to C++ and ROOT

M. van Leeuwen

February 7, 2013

C++ and ROOT

- C++ is a programming language
- ROOT is a collection of useful C++ code

  e.g.: Histograms, graphs, ntuples (lists) en file I/O
- AliRoot is an extension of ROOT with ALICE-specific code

  Detector geometry, track reconstruction, detector simulation, event generators, ...
- ROOT uses a C++ *interpreter* as *command line interface*

Compiler vs interpreter

A program can only be executed by a computer after it has been converted into machine instructions:

- *Compiler*: the whole program is converted into *executable*; can be run multiple times
  This is the normal way for C++
- *Interpreter*: read code line by line, convert and execute
  - ROOT interface works this way
  - Convenient for quick writing and testing of code
  - Can be slow for excecution of complex code
  - Other interpreted languages: java, VBScript, perl, python

Exercise 1

Create a file with the following content:

simple.C

```
{
  // Very simple test program
  for (int i = 0; i < 10; i++) {
    cout << "i " << i << endl;
  }
}
```

(Text editor: use Emacs or nedit)
and run it from the ROOT command line (`.x simple.C`)

Variables: declaration and initialisation

- `int i;`
  declaration: *i is an integer*
- `int i = 0;`
  declaration and initialisation: *i is an integer; set i to 0*
- `TH1F *h1;`
  declaration: *h1 is a pointer to a TH1F object*
- `TH1F *h1 = 0;`
  declaration and initialisation: *h1 is ...; set h1 to 0*

Common variable types:

- integer: int, Int_t, long int, Long64_t
- floating point number: float, Float_t, double, Double_t
- character (strings): char, Char_t, TString

The names with '_t' are platform-independent types defined in/by ROOT

Exercise 2

Extend the file

simple.C

```
{
  TH1F *histo = new TH1F("histo","test histogram",10,0,1);
  for (int i = 0; i < 10; i++) {
    float r = gRandom->Rndm();
    cout << "i " << i << " r " << r << endl;
    histo->Fill(r);
  }
  histo->Draw();
}
```

and run it from the ROOT command line (`.x simple.C`)

Objects and classes

A *Class* or *Object* is combination of data and algorithms:

- **Data members**
  Often not directly accessible; use Getters instead, e.g. `hist->GetXaxis()`
- **Member functions**
  Most can be called by used, e.g. `hist->Draw()` or `hist->Fill()`

e.g.: the histogram class TH1F
Formally:

- *Class* is the defintion of a type (e.g.: TH1F)
- *Object* is the instance of a class (i.e. a variabele, e.g.: hist)

ROOT files

ROOT provides a way to store objects in a file.

simple.C

```
{
  TFile *fout = new TFile("myhist.root", "RECREATE");
  TH1F *histo = new TH1F("histo", "test histogram",10,0,1);
  for (int i = 0; i < 10; i++) {
    float r = gRandom->Rndm();
    cout << "i " << i << " r " << r << endl;
    histo->Fill(r);
  }
  histo->Draw();
  fout->Write();
}
```

after execution, there is a file `myhist.root`

open with: `root myhist.root`

`new TBrowser`

Instantiating an Object

Creating an *instance* of a class in memory is *instantiation*

Two ways to instantiate an object in C++:

- `TH1F hist("hist", "my hist", 10, 0, 1)`
  - Makes a local variable that contains the object
  - Object is discarded when the variable goes *out of scope* (on the next closing accolade '}')
  - Call member functions with dot operator: `hist.Fill(1)`
- `TH1F *hist = new TH1F("hist", "my hist", 10, 0, 1)`
  - Allocates the object on the *heap* and makes a pointer to it
  - Object not discarded when `delete hist;` is called
  - Call member functions with arrow operator: `hist->Fill(1)`

It is commons to have multiple instances of the same class, e.g. different histograms

Exercise 3: a simple Monte Carlo

- Extend/change `simple.C` to add 2, 5, 10, 50 random numbers and fill the histogram
- What does the distribution look like?
- Fit the distribution

Exercise 4: a simple decay generator

1. Download gen_eta_phi.C from the website
2. Run it and look at the output tree
   ```
   dec_tree->Print(), dec_tree->Show(0);
   dec_tree->Draw("px"),
   dec_tree->Draw("sqrt(px*px+py*py)")
   ```
3. Now add decays $\pi^0 \to \gamma\gamma$ and inspect the result

How to generate a decay

1. Start in the restframe of the decaying particle
   1. The 2 decay products have equal and opposite momentum $p = \frac{1}{2}m$
   2. Direction: uniform in $\cos\theta$ and ϕ
2. Boost to lab frame
   - For a single axis (take line of flight):

$$\begin{aligned} p'_L &= \gamma p_L - \beta\gamma E \\ E' &= \gamma E - \beta\gamma p_L \end{aligned}$$

3. Store E, p_x, p_y, p_z for both decay photons in the output tree
4. Some things to look at:
   - Compare pt distribution of π^0 and decay particles
   - Energy asymmetry: $\alpha = \frac{E_2 - E_1}{E_1 + E_2}$
   - E_1 vs E_2 for bins in E

Appendix: C++ syntax

Common C++ syntax:

- `{ block }`
  Block of code
- `for ( init; cond; incr ) { }`
  Loop with counter
- `if ( expr ) { } else { }`
  Condition execution
- `while ( expr ) {}`
  Conditional loop (executed as long as expr is true)
- `do  while ( expr );`
  Conditional loop (executed at least once; until expr is fals)
- `cout << expr << endl;`
  print expr on screen (see also printf)

Appendix: Useful ROOT classes

Commonly used ROOT classes:

- `TH1F, TH2F, TH3F`
  histograms
- `TGraph, TGraphErrors`
  graphs (collection of points x,y)
- `TStyle/gStyle`
  graphics settings for drawing
- `TCanvas`
  window for drawing graphs/histograms
- `TFile`
  root file
- `TTree`
  Tree-object; (structured) list of data

refer to the ROOT website and User Guide for explanations

Compiled code: libraries and executables

Two types of compiled files:

- executables

  Run by double-click or type the name on the shell command line
  Windows: .exe files

  Linux: executable bit set (use '`ls -l`' to see, or '`file <filename>`')

- libraries

  Compiled collection of routines/algorithms; used in conjuction (loaded by) executables
  Most of the root code is packaged as libraries, e.g. libHist.so