Top 10 C++ Tips

Benno List

H1 OO Analysis Forum

8.12.2008

Introduction

- KISS: Keep It Simple, Stupid!
- The following tips are for analysis grade software,
  contributions to official H1OO software need more stringent rules
- Analysis software:
  Aim is a physics result, not the most beautiful design!
  => Follow the KISS principle: “Keep It Simple, Stupid!”
- But: your software must be written well enough to be
  - stable and reliable (run over millions of events!)
  - understandable (to you!) -> are you really sure what you are plotting?
  - flexible (change of cuts, binning, systematic error evaluation)
- My examples are untested and somewhat abbreviated (missing header files etc)

Tip 0: Learn a Bit of C++

- You know enough C++ to get your code running.
- If you now look *again* into your C++ book:
  - you'll understand more (“what does 'static' mean?” “what is 'const'?”)
  - you'll learn some tricks (formatted output with cout, string handling in C++) that you can use
- C++ is perhaps the most complicated programming language that exists, much more difficult than FORTRAN, but:
  - Good programmers can program FORTRAN in any language! :-)
  - A lot of things you'll probably never need (to write yourself):
    - operator overloading
    - templates
    - inheritance
  - => without these things, even C++ is manageable

Tip 1: Collect Constants in one File

```
if (Q2 >= 5 & Q2 < 100) {
  if (pt >= 3.5 && pt < 4.5)
    h_ptbin1->Fill (W);
  else if (pt >= 5 && pt < 6)
    h_ptbin2->Fill (W);
  else if (pt >= 6)
    h_ptbin3->Fill (W);
}
```

Bad: very hard to change number of bins or bin boundaries. Leads to bugs. Magic values hard to understand.

```
// MyConstants.h:
static const int nptbins = 3;
static const double ptbins[nptbins+1] = {3.5, 4.5, 6., 1E10};
static const double Q2mincut = 5, Q2maxcut=100;
```

"`static`" avoids compiler errors if file is included in different .C files.

```
#include "MyConstants.h"

if (Q2 >= Q2mincut && Q2 < Q2maxcut)
  for (int i = 0; i < nptbins; ++i)
    if (pt >= ptbins[i] && pt < ptbins[i+1])
      h_ptbin[i]->Fill (W);
```

Better: all numbers are collected in one place.

Tip 2: Use Functions

- Avoid 2000 line main programs in a single file
- If you copy-and-past code: write a function instead!
  => you'll have to fix the bugs only in one place!

```
// MyFunctions.h:
class TH1;
void Efficiency (TH1 *h1, TH1 *h2, TH1 *h3);
```

A very simple function to calculate the efficiency from two histograms with correct errors in case of weighted events (h3 = h1/h2)

```
// MyFunctions.C:
#include “MyFunctions.h”
#include <cmath>
using namespace std;
void Efficiency (TH1 *h1, TH1 *h2, TH1 *h3) {
  for (int i = 0; i <= h1.GetNbinsX()+1; ++i) {
    double sumw_1 = h1->GetBinContent(i);
    double sumw2_1 = pow (h1->GetBinError(i), 2);
    double sumw_2 = h2->GetBinContent(i);
    double sumw2_2 = pow (h2->GetBinError(i), 2);
    double eff = sumw_1/sumw_2;
    double err = sqrt (sumw2_1*pow (sumw_2-sumw_1, 2) +
                       (sumw2_2-sumw2_1)*pow (sumw_1, 2))/
                       pow (sumw2_2, 2);
    h3->SetBinContent (i, eff);
    h3->SetBinError (i, err);
  }
}
```

Tip 3: Use (Simple) Classes

```
// EventData.h:
class EventData {
  public:
    void Fill();
    double x, Q2;
};
```

"Fill" called for every event;
public access to event data, i.e. x, Q2

```
// EventData.C:
#include "EventData.h"

void EventData::Fill() {
  static H1FloatPtr yHat ("Ys");
  static H1FloatPtr Q2Hat ("Q2s");
  x = (*Q2Hat)/(*yHat * 4*27.55*920);
  Q2 = (*Q2Hat);
}
```

Fill() gets the data from HAT/mODS/ODS;
easy to change from Sigma method to Electron method, or introduce scales (systematics!)

```
//main.C:
#include "EventData.h"

EventData ed;
TH1F *hx = new TH1F("xBj", "Bjorken x", 100, 0, 1);
while (H1Tree::Instance()->Next()) {
  ed.Fill();
  hx->Fill (ed.x);
}
```

Now everywhere we need Bjorken x, we access the EventData class
-> easier to keep x, Q2 consistent (systematics!)

Simple Classes: 2nd Example

```
// EventLoop.h:
class EventData;
class EventLoop {
  public:
    EventLoop(EventData& ed_);
    void Fill();
    void Output();
    EventData& ed;
    TH1F *hx;
};
```

"Fill" called for every event;
Histos are booked in constructor,
filled with Fill(),
written out with Output()
EventData is accessed via an EventData reference

```
// EventLoop.C:
#include "EventLoop.h"
#include "EventData.h"
EventLoop::EventLoop (EventData& ed_)
: ed (ed_) {
  hx = new TH1F ("xBj", "Bjorken x", 100, 0, 1); }
void EventLoop::Fill() {
  hx->Fill (ed.x); }
void EventLoop::Output() {
  hx->Write(); }
```

This is how to initialize the reference

Histogram is booked,
filled,
and written out

```
//main.C:
#include "EventData.h"
#include "EventLoop.h"
EventData ed;
EventLoop loop (ed);
while (H1Tree::Instance()->Next()) {
  ed.Fill();
  loop.Fill();
}
Tfile outfile ("output.root", "RECREATE");
loop.Output();
```

Write separate classes for
- Main analysis
- Control plots
- Trigger efficiencies
- etc ...

Tip 4: Use Tstring (or std::string)

Bad: very hard to change number of bins or bin boundaries. Leads to bugs.
Magic values hard to understand.

```
TH1F h_ptbin1 = new TH1F (“ptbin1”, “W, 3.5<=pt<4.5”, 100, 0, 200);
TH1F h_ptbin2 = new TH1F (“ptbin2”, “W, 5<=pt<6”, 100, 0, 200);
TH1F h_ptbin3 = new TH1F (“ptbin2”, “W, pt>=6”, 100, 0, 200);
```

Better: all numbers are collected in one place.
Flexible, extensible

```
#include “MyConstants.h”
#include <TString>

TH1F *h_ptbin[nptbins];

for (int i = 0; i < nptbins; ++i) {
  Tstring id (“ptbin”);
  ptbin+=(i+1);
  Tstring title (“W, “);
  title += ptbins[i] += “<=pt<” += ptbins[i+1];
  h_ptbin[i] = new TH1F (id, title, nWbins, Wmin, Wmax);
}
```

Tip 5: Store only Numbers in RooT Trees

- RooT offers to store class objects in RooT trees
- Looks nice, but:
  - RooT Trees become unusable every time you change the class, i.e. all the time
  - Difficult to look at the RooT Trees interactively (e.g. with Tbrowser)
  - In 99.5% of all cases: you're better off with plain numbers
    => store px, py, pz, E of your objects instead of a TlorentzVector or your own class object

Tip 6: Be Aware (not Afraid) of Memory Leaks

- Every time you create an object with "`new`" and you have no corresponding "`delete`", you create a memory leak
- Writing code without memory leaks is very difficult in C++
- But: Memory leaks are ***not*** always a problem!
  -> Memory is reclaimed by the operating system after program stops.
- Just make sure that you don't run out of memory
- A program creating 100000 objects once (at the beginning) will run fine!
- A program creating 1 object in an event loop will
  - run fine on a test sample (of 10k events)
  - will always crash during your real job, running over 100M events!
- Tip: write out a message (`cout << "MyRoutine: Calling new\n";`) everytime you call `new`. If your output file becomes large, you have a problem.

Tip 7: Use (vector<> for) Collections

```
// main.C:
#include <vector>
using namespace std;

int main () {
  vector<TH1 *> myHistograms;
  TH1F *h1 = new TH1F (“xBj”, “Bjorken x”, 100, 0, 1);
  myHistograms.push_back (h1);
  TH1F *h2 = new TH2F (“xBjQ2”, “Bjorken x vs Q2”, 100, 0, 1, 100, 0, 10000);
  myHistograms.push_back (h2);

  // fill histograms

  Tfile (“out.root”, “RECREATE”);
  for (unsigned int i = 0; i < myHistograms.size(); ++i) {
    myHistograms[i]->SetLineColor (2);
    myHistograms[i]->SetLineWidth (3);
    myHistograms[i]->Write();
  }
}
```

without this, you have to write std::vector

Example here stores pointers to histograms
(remember: all RooT histo classes are subclasses of TH1)

- It is often convenient to have an array of (pointers to) many objects (histograms, numbers etc)
- vector<Type> allows easy storage, retrieval by number, knows always how many objects are stored

Tip 8: Use Command Line Arguments

```
// main.C:
#include <H1Steering/H1CmdLine.h>
#include <H1Steering/H1OptionString.h>
#include <H1Steering/H1OptionInt.h>

int main (int argc, const char *argv[]) {
  H1OptionString outfilename (“out.root”);
  H1OptionString systematics (0);
  H1CmdLine opts;
  opts.AddOption (“outfile”, 'o', outfilename);
  opts.AddOption (“systematics”, 's', systematics);
  opts.Parse (&argc, argv);

  Tfile outfile (outfilename);
  if (systematics == 0) {
    // do nominal analysis
  }
  else if (systematics == 0) {
    // do systematics 1
  }
  return 0;
}
```

You have 1 executable,
which can be run several times
(in parallel!)
with different settings.

No recompiling needed!
=> send many jobs to the farm to do your analysis

Tip 9: (Learn to) Use Text Files

```
// efficiencies.txt
5
0 100
80.3  85.5  87.2  92.1  83.1
```

Efficiency file,
probably written by some other part
of your program
(efficiencies in percent)

```
// main.C
#include <fstream>
int main () {
  ifstream eff_file (“efficiencies.txt”);
  int n;
  double xmin, xmax, eff;
  eff_file >> n >> xmin >> xmax;
  TH1F *h_eff = new TH1F (“eff”, “Efficiencies”, n, xmin, xmax);
  for (int i = 0; i<n; ++i) {
    eff_file >> eff;
    h_eff->SetBinContent (i+1, 0.01*eff);
  }
  ofstream out_file (“table_eff.tex”);
  out_file << “\\begin{tabular}{ccc}\n”
           << “$x_{min}$ & $x_{max}$ & $\epsilon$ \\\\\n\\hline\n”;
  for (int i = 1; i <= h_eff->GetNbinsX(); ++i) {
    out_file << “$” <<h_eff->GetBinLowEdge (i) << “$ & $“
             << h_eff->GetBinLowEdge (i+1) << “$ & $“
             << 100*h_eff->GetBinContent (i) << “\,\\%$\\\\\n“;
  }
  out_file << “\\end{tabular}\n”;
}
```

Reads in efficiency file,
books and fills a histogram from it

Writes the contents of a histogram
into a TEX table, ready for inclusion
in your thesis!

two slashes needed for
one output slash!

Tip 10: Always Write out Root Histograms

- It is very conventient to write out a postscript file directly from your analysis job
- But remember: for your thesis / H1preliminary / publication you'll need (more) fancy formatting
- Therefore: always write out all your RooT histograms into a RooT file that you can later use to produce nice plots without re-running the analysis job!

Conclusions

- For analysis purposes, you don't need many fancy C++ features
- But knowing a bit more about C++ helps to improve
  - your code
  - your analysis!
- The crisis often comes when you have written your selection code and start to do systematics
  -> this may be a good time to re-work some of your analysis code