Universität Zürich[UZH]
Physik-Institut

Lecture 9: Programming using ROOT 5 and 6 packages

1 Introduction

While ROOT scripting and command lines are powerful tools, they are not without flaws. You can crash ROOT relatively easily, you cannot parallel process instances of ROOT process executions and if there is an error due to a bug, the bug may be in your code and/or in ROOT source code. Furthermore, a script may work with one version of ROOT, but may not, or have a completely different result in another version. This is because like anything else, ROOT is a large project with many different contributors. Same thing for C++ libraries.
However, we don't have to worry too much about the bugs within C++ packages since most of the packages we use have been tested by programmers for many decades. Furthermore, we can choose which packages to use and which packages not to use. The new packages under development are typically unstable and we can simply choose not to compile our code with these unstable packages. Similarly, instead of using ROOT entirely, you can instead take advantage of what ROOT offers by compiling your program with specific ROOT libraries, objects and functions. This way, you can have full functionality of ROOT but also with stability and performance of a regular C++ program.
Today, we will talk about programming using these packages.

1.1 Compiling with ROOT packages

Before we go on programming using ROOT packages, let's make sure we know how to compile using ROOT packages. Before compiling, you must first load ROOT PATHs[1]. Recall

```
source /app/cern/root_v6.18.04/bin/thisroot.sh
```

Now we can compile using the ROOT packages with a built-in flag called root-config.

```
> root-config
Usage: root-config [--prefix[=DIR]] [--exec-prefix[=DIR]] [--version]
 ↪  [--cflags] [--auxcflags] [--ldflags] [--new] [--nonew] [--libs]
 ↪  [--glibs] [--evelibs] [--bindir] [--libdir] [--incdir] [--etcdir]
 ↪  [--tutdir] [--srcdir] [--noauxcflags] [--noauxlibs] [--noldflags]
 ↪  [--has-<feature>] [--arch] [--platform] [--config] [--features] [--ncpu]
 ↪  [--git-revision] [--python-version] [--cc] [--cxx] [--f77] [--ld ]
 ↪  [--help]
```

As you can see, there are many options. For today we will stick with the configuration

```
root-config --cflags --glibs
```

You are free to try other flags also. In order to implement this, you need to declare a new variable in your Makefile as the following.

[1] PATH and LD_LIBRARY_PATH

```
#This is the directory of YOUR source code.
sourcedirectory=./

#These are your source codes and components
trial=$(wildcard *.cxx)
first_part=myobjects.cxx
second_part=main.cxx
component=myobjects.cxx

#Here, we're defining the compilers.
CC=gcc
CPP=g++
NVCC=nvcc

#We're defining systems variable such as "remove" from system and
↪  "timestamp"
RM=rm
TIMESTAMP=$(shell date +"%Y_%m_%d_T-%H_%M" )

SFLAG=-Wall
PFLAG=-lncurses
ROOTFLAG=-g $(shell root-config --cflags --glibs)

ALLFLAG= $(SFLAG) $(ROOTFLAG)


objects = $(trial:.cxx=)

#objects =
#When a Makefile is executed, by default it tries the option "all"
all: clean $(objects)
#We will tell the makefile to clean, compile the first component, second
↪  then the third component.


$(objects): %: %.cxx

        @echo Compiling $(sourcedirectory)$<
        @$(CPP) -o $(addprefix run_,$@.exe) $(ALLFLAG) $< $(PFLAG)
        @echo Successfully compiled $(sourcedirectory)$<
        @echo executable is $(addprefix run_,$@.exe)

first:
#Here, we define what "first_one" will do.
#At sign @ will silence the command appering in the terminal
        @echo Compiling $(sourcedirectory)$(first_part)
        @$(CPP) -o $(first_part).o  $(SFLAG) $(sourcedirectory)$(first_part)
        ↪  $(PFLAG)
        @echo Successfully compiled $(sourcedirectory)$(first_part)
        @echo The compiled object is $(first_part).o
second:
#Let's now compile the second part.
        @echo Compiling $(sourcedirectory)$(second_part)
```

```
51          @$(CPP) -c -o $(second_part).o  $(SFLAG)
            ↪  $(sourcedirectory)$(second_part) $(PFLAG)
52          @echo Successfully compiled $(sourcedirectory)$(second_part)
53          @echo The unliked compiled code is $(second_part).o
54
55          @echo Compiling $(sourcedirectory)$(component)
56          @$(CPP) -c -o $(component).o $(SFLAG) $(sourcedirectory)$(component)
            ↪  $(PFLAG)
57          @echo Successfully compiled $(sourcedirectory)$(component)
58          @echo The unliked compiled code is $(component).o
59  third:
60  #Let's link the second part
61          @echo Linking $(component).o and $(second_part).o
62          @$(CPP) -o compiled_program.exe $(SFLAG)
            ↪  $(sourcedirectory)$(component).o
            ↪  $(sourcedirectory)$(second_part).o $(PFLAG)
63          @echo Successfully compiled $(sourcedirectory)$(component)
64          @echo Everything is linked and compiled into compiled_program.exe
65  clean:
66          @echo $(TIMESTAMP)
67          @echo "Making old/$(TIMESTAMP) directory"
68          $(shell mkdir -p old/$(TIMESTAMP) )
69          @echo "Copying the source to the old directory"
70          $(shell cp -r $(sourcedirectory)/*.cxx old/$(TIMESTAMP) )
71          @echo "Moving all .exe to the old directory"
72          $(shell mv *.exe old/$(TIMESTAMP) )
73          $(shell mv *.o old/$(TIMESTAMP) )
74          @echo "Copying the Makefile to the old directory"
75          $(shell cp Makefile old/$(TIMESTAMP) )
76  #        £ (a dummy line just placed this to exit mathmode in LATEX)
```

You need to particularly pay attention to line 21 where the "ROOTFLAG" is defined. Also, line 23 where "ALLFLAG" are defined. Finally note that line 37 "ALLFLAG" is used instead of "SFLAG".
You can also define ROOTFLAG in a "manual" method without having to set source thisroot.sh by using the following compilation flag

```
1  reduced_ROOTFLAG=-g -m64 -I/cern/root_v6.18.04/include
   ↪  -L/cern/root_v6.18.04/lib -lGui -lCore -lCint -lRIO -lNet -lHist -lGraf
   ↪  -lGraf3d -lGpad -lTree -lRint -lPostscript -lMatrix -lPhysics -lMathCore
   ↪  -lThread -lm -ldl
```

How you do this is entirely up to you. The important part is that you get all of the packages necessary for your task and that it compiles and executes without any issues.

1.2 Include ROOT packages

Now that we know how to compile using ROOT packages, let's include them in our C++ code. Inspect the following skeleton code as you will have to write something similar in your first practical session.

```
1  //root_code.cxx
2  #include <stdio.h>
3  #include <vector>
```

```
4  #include <malloc.h>
5  #include <fstream>
6  #include <stdlib.h>
7  #include <string.h>
8  #include <stdio.h>
9  #include <ctype.h>
10 #include <iostream>
11 #include <vector>
12 #include <cmath>
13 #include <sstream>
14 #include <cfitsio/fitsio.h>
15 //ROOT Packages
16 #include<TStyle.h>
17 #include<TFile.h>
18 #include<TTree.h>
19 #include<TH2.h>
20 #include<TH1.h>
21 #include<TCanvas.h>
22 #include<Riostream.h>
23 #include<TROOT.h>
24 #include<TClass.h>
25 #include<TMath.h>
26 #include<THashList.h>
27 #include<TF2.h>
28 #include<TF3.h>
29 #include<TPluginManager.h>
30 #include<TVirtualPad.h>
31 #include<TRandom.h>
32 #include <TVirtualFitter.h>
33 #include <THLimitsFinder.h>
34 #include <TProfile.h>
35 #include <TStyle.h>
36 #include <TVectorF.h>
37 #include <TVectorD.h>
38 #include <TBrowser.h>
39 #include <TObjString.h>
40 #include <TError.h>
41 #include <TVirtualHistPainter.h>
42 #include <TVirtualFFT.h>
43 #include <TSystem.h>
44 #include <HFitInterface.h>
45 #include <Fit/DataRange.h>
46 #include <Fit/BinData.h>
47 #include <Math/MinimizerOptions.h>
48 #include <Math/QuantFuncMathCore.h>
49 #include <TCanvas.h>
50 #include <TString.h>
51 #include <TLatex.h>
52 #include <TRandom3.h>
53 #include <TH2F.h>
54 #include <TF1.h>
55 #include <TError.h>
56 #include <TASImage.h>
```

```
#include <TAttImage.h>
#include <TImage.h>
#include <TThread.h>
#include <TLine.h>
#include <TColor.h>
#include <TPaletteAxis.h>
#include <TFITS.h>
#include <TVirtualFFT.h>

#include<iostream>
#include<stdio.h>
#include<vector>
#include<fstream>
#include<sstream>

using namespace std;
int main(int argc,char *argv[]){//Main begins
        TRandom3* randomer=new TRandom3();
        TFile * file= new TFile("myrootfile.root", "RECREATE");
        TH1I * integerhist=new TH1I("integerhist","integerhist",10,0,100);
        TCanvas * mycanvas=new TCanvas("mycanvas","mycanvas", 1920,1080);

        for (int i=0;i<1000;i++){
                integerhist->Fill( randomer->Gaus(50,10) );
        }

        mycanvas->cd();
        integerhist->Draw();
        mycanvas->Print("myTH1I.png");

        integerhist->Fit("gaus");
        integerhist->Draw();
        mycanvas->Print("myTH1I_fitted.png");

        file->cd();
        integerhist->Write();
    file->Close();
        delete file;
    return 0;
}//Main Ends
```

As you can see above, these are many ROOT packages. Some you may need, some not. If you want to inspect what's available in your ROOT installation. You can check the "include" directory in your ROOT installation directory. For instance, from spinorXX.physik.uzh.ch

```
ls /app/cern/root_v6.18.04/include
```

For now, we will continue into our practical session and feel free to experiment to include more or less packages than the included packages shown in the skeleton above.

= The practical programming part of this course will now begin for 60 minutes. =

2 Compiling and using the skeleton code

1. Make modifications to your Makefile to compile using ROOT packages.
2. Write a similar skeleton code as the last example and compile.
3. Include your functions and objects and compile everything together and with root packages.
4. Create a new header and a accompanying C++ file called "myrootfunctions".
5. in your "myrootfunctions" be sure to have the following:
   - A function to plot a 1D histogram given a stack array.
   - A function to plot a 1D histogram given a heap array.
   - A function to plot a 1D histogram given a stack vector.
   - A function to plot a 1D histogram given a heap vector.
   - A function to plot a 2D histogram given a 2D stack array.
   - A function to plot a 2D histogram given a 2D heap array.
   - A function that takes a TH1 and Draws on a TCanvas and prints the TCanvas.
   - A function that takes a TH1 and Writes it into a root file.
   - Same as the last two, except for TH2 drawing using "colz" option.
   - A function that takes a 1D histogram, and extracts all of its elements into an array in stack and heap.
   - A function that takes a 1D histogram, and extracts all of its elements into an vector in stack and heap.
6. Test and make sure all functions in your functions, object and rootfunctions packages compile and work by sampling each and every one of them your main.

= The theoretical lecture part of this course will now continue for 15 minutes. =

3 Questions and catch up time

In today's first part of the lecture and practical session have covered almost everything we need to. We will not cover further material for ROOT, but this is your chance to ask questions and catch up on everything up until this point. The next practical session will help you exercise performing data analysis using ROOT packages.

= The practical programming part of this course will now begin for 60 minutes. =

4 More practice

1. If you have not, complete the tasks assigned from the last practical session.
2. Generate an array and a vector of 10000 random integers.
3. Plot the array and vector each into TH1I and perform the most appropriate fit.
4. Assemble the array and the vector into a TH2I where array elements are the X values corresponding to vector elements defined as the Y values.
5. Plot the above with the "colz" option.
6. Calculate how many integers you can store in 0.5 GB.
7. Create a 0.5 GB integer array and a 0.5 GB vector and fill them with random numbers.
8. Plot the arrays and vectors each into TH1I.
9. Super-impose them in TH1I using "same" option in draw.
10. Be sure to show the difference between the array and vector in the same TH1I by using two different colours. Use the built-in ->SetLineColor(kRed)/->SetLineColor(kBlue) for the TH1Is.
11. Fit each histogram with appropriate fits.
12. Be sure to show your histograms and fits all in one TCanvas and print it into a file as a png.
13. Similarly as before, plot the array/vector into a TH2I.
14. Save everything in your .rootfile.
15. inspect everything using TBrowser.

5 conclusion

At this point, you should have all knowledge required to perform any basic and intermediate data analysis using C++ and ROOT. The only thing we lack now is that we have not learned to fully utilize the computer. We know how to use all of its RAM, its hard drive, but we haven't had very much control over all of the CPUs.
Since the 2000s, multi-core and multi-thread CPUs have become very popular. Now days, even our phones have multi-threaded multi-core CPUs. Up until now, we allowed the computer to decide how to use its CPU to execute our program. In the next class, we will cover parallel programming and we will go over how to use all of our computer.