Lecture 9: Programming using ROOT 6 packages

1 Introduction to Programming with ROOT

While ROOT scripting and its command-line interface are powerful tools, they are not without their drawbacks. ROOT can crash relatively easily, it does not support running multiple instances in parallel, and debugging errors can be challenging because the issue may stem from your own code or from ROOT 's source code itself. Additionally, a script that works with one version of ROOT may fail or produce entirely different results with another version. This is not surprising, as ROOT is a large project with many contributors. The same considerations apply to other C++ libraries as well.

Fortunately, we do not need to worry too much about bugs in well-established C++ packages. Most of the libraries we commonly use have been extensively tested and refined over decades. We also have the flexibility to choose which libraries to include in our projects, avoiding unstable or experimental packages that are still under development. Similarly, rather than relying entirely on ROOT scripting, we can instead integrate ROOT functionality into our compiled C++ programs by selectively using specific ROOT libraries, objects, and functions. This approach allows us to enjoy the full feature set of ROOT while maintaining the stability, performance, and reliability of a compiled C++ program.

Today, we will focus on programming with these ROOT packages.

1.1 Compiling with ROOT packages

Before we begin programming with ROOT packages, let's first ensure we know how to compile code using ROOT. Before compiling, you need to set up the ROOT environment paths (specifically, PATH and LD_LIBRARY_PATH). If you have ROOT on your local machine, make sure you have run (as in the previous lecture):

```
source <your_root_installation_folder>/bin/thisroot.sh
```

or, if you are working on machines of the Physics Institute's cluster:

```
source /app/cern/root_v6.36.10/bin/thisroot.sh
```

Now we can compile code using ROOT packages by utilizing the built-in tool `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] [--python3-version] [--cxxstandard] [--cc] [--cxx]
↪  [--f77] [--ld ] [--help]
```

There are many available configuration options, but here we only need the necessary compiler and linker flags: `--cflags`, which provides the required include paths for header files, and `--glibs`, which provides the library paths and libraries required for linking:

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

returns (on a machine of the Physics Institute's cluster):

```
-pthread -std=c++17 -m64 -I/app/cern/root_v6.36.10/include -L/app/cern/root_v6.36.10/lib
↪  -lGui -lCore -lImt -lRIO -lNet -lHist -lGraf -lGraf3d -lGpad -lROOTVecOps -lTree
↪  -lTreePlayer -lRint -lPostscript -lMatrix -lPhysics -lMathCore -lThread -lROOTNTuple
↪  -lMultiProc -lROOTDataFrame -lROOTNTupleUtil -Wl,-rpath,/app/cern/root_v6.36.10/lib
↪  -pthread -lm -ldl -rdynamic
```

If you run the same command on your local machine, `appcernroot_v6.36.10` will be replaced by `<your_root_installation_folder>`.

To use this information to compile our `C++` code using `ROOT`, we define a new variable in our `Makefile`, as shown below:

```
# This is the directory of YOUR source code
sourcedirectory=./
#These are your source codes and components
myfunctions=myfunctions.cxx
myobjects=myobjects.cxx
mymain=root_code.cxx
output=$(mymain:.cxx=)
# Here, we're defining the compilers
CC=gcc
CPP=g++
NVCC=nvcc
# We're defining systems variable such as "remove" from system and "timestamp"
RM=rm
TIMESTAMP=$(shell date +"%Y_%m_%d_T-%H_%M_%S" )
SFLAG=-Wall
ROOTFLAG=-g $(shell root-config --cflags --glibs)
ALLFLAG= $(SFLAG) $(ROOTFLAG)
# When a Makefile is executed, by default it tries the option "all"
all: clean functions objects main
# We tell the makefile to clean, then to compile the functions and objects, and eventually
↪  to compile main and to link it with the other components.
functions:
# Compile the functions part
        @echo Compiling $(sourcedirectory)$(myfunctions:.cxx=.o)
        $(CPP) -c -o $(myfunctions:.cxx=.o) $(ALLFLAG) $(sourcedirectory)$(myfunctions)
        @echo Successfully compiled $(sourcedirectory)$(myfunctions)
        @echo The unlinked compiled code is $(myfunctions:.cxx=.o)
objects:
# Compile the objects part
        @echo Compiling $(sourcedirectory)$(myobjects)
        $(CPP) -c -o $(myobjects:.cxx=.o) $(ALLFLAG) $(sourcedirectory)$(myobjects)
        @echo Successfully compiled $(sourcedirectory)$(myobjects)
        @echo The unlinked compiled code is $(myobjects:.cxx=.o)
main:
# Compile main and link it with functions and objects
        @echo Linking $(myobjects:.cxx=.o) and $(myfunctions:.cxx=.o) with $(mymain)
        $(CPP) -o $(addprefix run_,$(output).exe) $(ALLFLAG)
        ↪  $(sourcedirectory)$(myobjects:.cxx=.o) $(sourcedirectory)$(myfunctions:.cxx=.o)
        ↪  $(mymain)
        @echo Successfully compiled $(sourcedirectory)$(output)
        @echo Everything is linked and compiled into $(addprefix run_,$(output).exe)
clean:
        @echo $(TIMESTAMP)
        @echo "Making old/$(TIMESTAMP) directory"
        $(shell mkdir -p old/$(TIMESTAMP) )
        @echo "Copying the source to the old directory"
        $(shell cp -r $(sourcedirectory)/*.cxx old/$(TIMESTAMP) )
        $(shell cp -r $(sourcedirectory)/*.h old/$(TIMESTAMP) )
```

```
46          @echo "Moving all .exe to the old directory"
47          $(shell mv *.exe old/$(TIMESTAMP) )
48          $(shell mv *.o old/$(TIMESTAMP) )
49          @echo "Copying the Makefile to the old directory"
50          $(shell cp Makefile old/$(TIMESTAMP) )
```

Particular attention should be paid to line 16, where `ROOTFLAG` is defined, line 17, where `ALLFLAG` is defined, and lines 24, 30, and 36, where `ALLFLAG` is used instead of `SFLAG` in previous examples.

Alternatively, you can define `ROOTFLAG` manually — this would work even without sourcing `thisroot.sh` — by specifying the following compilation flags explicitly instead of line 16:

```
16 ROOTFLAG=-pthread -std=c++17 -m64 -I/app/cern/root_v6.36.10/include
   ↪ -L/app/cern/root_v6.36.10/lib -lGui -lCore -lImt -lRIO -lNet -lHist -lGraf -lGraf3d
   ↪ -lGpad -lROOTVecOps -lTree -lTreePlayer -lRint -lPostscript -lMatrix -lPhysics
   ↪ -lMathCore -lThread -lROOTNTuple -lMultiProc -lROOTDataFrame -lROOTNTupleUtil
   ↪ -Wl,-rpath,/app/cern/root_v6.36.10/lib -pthread -lm -ldl -rdynamic
```

Making use of the `root-config` tool, even the direct compilation command is sufficiently short:

```
g++ -o run_root_code.exe root_code.cxx myfunctions.cxx myobjects.cxx $(root-config --cflags
↪ --glibs)
```

Which compilation option you choose is entirely up to you. The key point is to ensure that all required headers are included, the corresponding libraries are linked to the program, and the required dynamic libraries are available at runtime.

1.2 Include ROOT packages

Now that we know how to compile code with `ROOT` packages, let's integrate them into our `C++` programs. The following example code provides a long list of headers of `ROOT` classes and functions, some of which you will need in the first practical session:

```
1  // root_code_headerlist.cxx
2  #include <iostream>
3  #include <fstream>
4  #include <sstream>
5  #include <vector>
6  #include <cmath>
7  #include <malloc.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <stdio.h>
11 #include <ctype.h>
12 // ROOT Packages
13 #include <TStyle.h>
14 #include <TFile.h>
15 #include <TTree.h>
16 #include <TH2.h>
17 #include <TH1.h>
18 #include <TCanvas.h>
19 #include <Riostream.h>
20 #include <TROOT.h>
21 #include <TClass.h>
22 #include <TMath.h>
23 #include <THashList.h>
24 #include <TF2.h>
25 #include <TF3.h>
26 #include <TPluginManager.h>
27 #include <TVirtualPad.h>
```

```
#include <TRandom.h>
#include <TVirtualFitter.h>
#include <THLimitsFinder.h>
#include <TProfile.h>
#include <TVectorF.h>
#include <TVectorD.h>
#include <TBrowser.h>
#include <TObjString.h>
#include <TError.h>
#include <TVirtualHistPainter.h>
#include <TVirtualFFT.h>
#include <TSystem.h>
#include <HFitInterface.h>
#include <Fit/DataRange.h>
#include <Fit/BinData.h>
#include <Math/MinimizerOptions.h>
#include <Math/QuantFuncMathCore.h>
#include <TString.h>
#include <TLatex.h>
#include <TRandom3.h>
#include <TH2F.h>
#include <TF1.h>
#include <TASImage.h>
#include <TAttImage.h>
#include <TImage.h>
#include <TThread.h>
#include <TLine.h>
#include <TColor.h>
#include <TPaletteAxis.h>
using namespace std;
int main(int argc, char *argv[]){
  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;
}
```

This list of ROOT headers is not necessarily exhaustive, but certainly highly redunant for the actual example. As previously, we only need to include the header files that are really required, so our example code can be significantly shortened:

```
// root_code.cxx
#include <TRandom3.h>
#include <TFile.h>
#include <TH1I.h>
#include <TCanvas.h>
int main(int argc, char *argv[]){
  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);
```

```
11    for (int i = 0; i < 1000; i++){
12      integerhist->Fill(randomer->Gaus(50, 10));
13    }
14    mycanvas->cd();
15    integerhist->Draw();
16    mycanvas->Print("myTH1I.png");
17    integerhist->Fit("gaus");
18    integerhist->Draw();
19    mycanvas->Print("myTH1I_fitted.png");
20    file->cd();
21    integerhist->Write();
22    file->Close();
23    delete file;
24    return 0;
25  }
```

There are many `ROOT` packages — some you may need, and some not. If you want to see what is available in your `ROOT` installation, you can inspect the `include` directory within your `ROOT` installation folder. For example, on `ohm01.physik.uzh.ch`, you can check:

```
ls /app/cern/root_v6.36.10/include
```

For now, we will proceed with the practical session. Feel free to experiment by including more or fewer packages than those shown in the skeleton code above.

We will now start a 60-minute practical programming session.

2 Practical Session – Part I

Compiling and using the skeleton code - Extend your Packages

1. Modify your `Makefile` so that it compiles using the `ROOT` packages.
2. Write a skeleton code similar to the example in the lecture and compile it.
3. Add your `myfunctions` and `myobjects`, and compile everything together with the `ROOT` packages.
4. Create a new header file and a corresponding `C++` source file called `myrootfunctions.h` and `myrootfunctions.cxx`, respectively.
5. In your `myrootfunctions`, ensure that you implement the following for integer numbers:
   (a) A function to plot a 1D histogram from a *stack* or *heap* `array`.
   (b) A function to plot a 1D histogram from a *stack* `vector`.
   (c) A function to plot a 1D histogram from a *heap* `vector`.
   (d) A function to plot a 2D histogram from a 2D *stack* `array`.
   (e) A function to plot a 2D histogram from a 2D *heap* `array`.
   (f) A function that takes a `TH1` and draws on a `TCanvas` and prints the `TCanvas`.
   (g) A function that takes a `TH1` and writes it into a `ROOT` file.
   (h) Versions of the above two functions for `TH2` drawing using the `colz` option.
   (i) A function that takes a 1D histogram, and extracts all of its elements into a *heap* `array` (Consider why a *stack* `array` does not work here in general!).
   (j) A function that takes a 1D histogram, and extracts all of its elements into `vector`s (both *stack* and *heap*).
6. Test all functions in your `myrootfunctions` package by calling each of them from your `main`, ensuring that they compile and work correctly. Also verify that the functions from `myfunctions` and `myobjects` still work when invoked from here.

We will now continue with a 15-minute theory session.

3 Questions and catch-up time

In today's first part of the lecture and practical session, we have covered almost everything we need to know about `ROOT`. We will not introduce additional material for `ROOT` at this time, but this is your opportunity to ask questions and review any concepts covered so far. In the next practical session, you will get hands-on experience performing data analysis using `ROOT` packages.

We will now start a 60-minute practical programming session.

4 Practical Session – Part II

More practice

1. Generate an `array` and a `vector` of 10000 random integers.
2. Plot the `array` and the `vector` each in a `TH1I` and perform the most appropriate fit.
3. Combine the `array` and the `vector` into a `TH2I`, where the `array` elements are the `X` values and the `vector` elements are the `Y` values.
4. Plot this histogram using the `colz` option.
5. Calculate how many integers can be stored in 0.5 GB of memory.
6. Create a 0.5 GB integer `array` and a 0.5 GB `vector` and fill them with random numbers.
7. Plot both the `array` and the `vector` each in a `TH1I`.
8. Superimpose them in a single `TH1I` using the `same` draw option. Make sure to distinguish between the `array` and the `vector` in the same `TH1I` by using two different colours. Use the built-in methods `->SetLineColor(kRed)` and `->SetLineColor(kBlue)` for the `TH1I` objects.
9. Fit each histogram with appropriate fits.
10. Display all histograms and fits together in a single `TCanvas` and export it as a `PNG` file.
11. As before, plot the `array` and `vector` in a `TH2I`.
12. Save everything in your `.root` file.
13. Inspect all contents using `TBrowser`.

We will now conclude with a 15-minute question session.

5 Conclusion

At this point, you should have all the knowledge required to perform basic and intermediate data analysis using `C++` and `ROOT`. The only area we have not yet explored is how to fully utilize the computer's processing power. We know how to make use of its RAM and storage, but we have not gained much control over its CPUs.
Since the early 2000s, multi-core and multi-threaded CPUs have become increasingly common; today, even smartphones are equipped with such processors. Up to now, we have allowed the computer to decide how to allocate CPU resources to execute our programs. In the next class, we will focus on parallel programming and learn how to fully leverage all of the computing power available.