COMPUTING METHODS IN HIGH ENERGY PHYSICS

S. Lehti

Helsinki Institute of Physics

Spring 2025

Combining languages

FORTRAN and C++

To access FORTRAN programs from C++ (C) and vice versa, look at e.g.
http://arnholm.org/software/cppf77/cppf77.htm
http://cnlart.web.cern.ch/cnlart/217/node34.html

Since new HEP programs are mostly written in C++, you may need to call FORTRAN subroutines from C++ main, but having FORTRAN main calling C++ is unlikely. Therefore we concentrate here only on the former case.

The FORTRAN code can be accessed from C++ via COMMON blocks. Let's consider a FORTRAN subroutine named TEST with the following common block:

```
INTEGER I
REAL R
DOUBLE PRECISION D
COMMON/COMX/I,R(3,3),D
CHARACTER*80 CHTEXT(10)
COMMON/COMC/CHTEXT
```

In C++, function or structure corresponding to fortran subroutine or common block has to be declared with the same name typed in lower case letters with underscore

```
SUBROUTINE TEST
test_;
```

For the C++/FORTRAN interface we need:

```
extern "C" void test_();
```

Combining languages

```
struct fBlockCOMX {
  int i;
  float r[3][3];
  double d;
};
extern "C" {
  extern fBlockCOMX comx_;
}
```

The common block variables can then be accessed as the structure member data fields

```
comx_.i = i;
float f = comx_.r[0][0];
```

Same applies to the character common block

```
struct fBlockCOMC {
  char text[10][80];
};
extern "C" {
  extern fBlockCOMC comc_;
}
```

Argument passing in calling FORTRAN from C++ is very compiler dependent. The following should work on most compilers

```
SUBROUTINE TEST(X,CH,Y)
float x,y;
char* ch_pointer;
int ch_length;
test_(&x,ch_pointer,&y,ch_length);
```

To avoid problems with argument passing, one could write a FORTRAN interface, which communicates with C++ using common blocks, and calls the actual subroutine which one wants to link

Combining languages

with the C++ program.

Compiling the C++/FORTRAN program: Since the FORTRAN code is probably something which is not modified, create a library containing the FORTRAN part and perhaps the interface, too. After all, this linkage procedure is practical only for reusing old existing FORTRAN code, when rewriting the FORTRAN code in C++ is too time consuming a task. The library needs to be compiled with a FORTRAN compiler. The formed library is used as any C++ library.

A C++/FORTRAN interface is found also in CMSSW, which is a reconstruction program and can use different event generators written in C++ or FORTRAN.

C++ and ROOT

C++ and ROOT are very close to each other and therefore easy to combine. The ROOT classes and libraries are available as soon as ROOT is installed in the system. In the Makefile one needs to include the ROOT include path

-I$(ROOTSYS)/include

and link the (used) ROOT libraries. On runtime the environment variable LD_LIBRARY_PATH must contain the path to the ROOT libraries.

Although linking C++ and ROOT is easy, why should one do such a thing? First of all, one can write small test programs with ROOT.

Combining languages

Secondly, in a typical physics analysis one has to analyse perhaps millions of events and speed becomes an issue. Executing ROOT macros is much slower than executing compiled code, so at some point compiling the ROOT analysis script may become relevant. Third reason is that language constructs not fully supported by CINT become available.

Example: let's consider a script using TGraph. How to compile that?
void myGraphPlotting(){ ... }
The first modification is to change "void myGraphPlotting" into "int main". Then the used class headers and namespace should be included
#include <iostream>
#include "TGraph.h"
#include "TCanvas.h"
#include "TH2F.h"
#include "TLatex.h"
using namespace std;

A Makefile should be added. The ROOT include path needs to be included, as well as a list of ROOT libraries

LIBS = -L$(ROOTSYS)/lib -lCint -lGpad -ldl
or
LIBS := $(shell root-config –glibs)

Notice that now the figure is not printed on screen, but only in a file.

Another way to compile the example is to use ACLiC, the AutomatiC Library Compiler for CINT. When

Combining languages

compiling the code into a library this way, no Makefile is needed, and the function name doesn't need to be changed. However, the header files need to be included. The script is compiled by typing

root [] .L myScript.C+

This + option creates a shared library myScript_C.so in your directory. After loading the library, the function becomes available

root [] myScript()

Now the ROOT command line is used, and the Canvas is available interactively.

To make your scripts to move easily between the interpreter and the compiler, it is recommended to always write the include statements in your scripts. Also do not use the CINT extensions and program around the CINT limitations.

Rootification of a class

- The class must inherit TObject
- The ClassDef(MyClass,1) macro is added in the class definition
- The ClassImp(MyClass) macro is added in the class implementation.

The TObject class provides the default behavior and protocol for the objects in the ROOT system. It is the primary interface to classes providing I/O, error handling, drawing etc.

Combining languages

Classes can be added in ROOT also without the ClassDef and ClassImp macros, but then the object I/O features of ROOT will not be available for these classes.

Note that you must provide a default constructor for your classes or you will get a compiler error. In order to get your rootified code to compile, a dictionary is needed. Dictionary is needed in order to get access to the classes via the interpreter. Dictionaries can be created with a *rootcint* program. The rootcint program also generates the

Streamer(),
TBuffer &operator>>() and
ShowMembers()

methods for ROOT classes, i.e. classes using the ClassDef and ClassImp macros.

To tell rootcint for which classes the method interface stubs should be generated, a file LinkDef.h is used. The LinkDef file name MUST contain the string: LinkDef.h or linkdef.h, e.g. MyLinkDef.h.

LinkDef.h must be the last argument on the rootcint command line.

A LinkDef file looks like the following:

```
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;
```

Combining languages

#pragma link C++ class MyJet+;
#pragma link C++ class vector<MyJet>;
#pragma link C++ class MyEvent+;

#endif

The trailing + tells rootcint to use the new I/O system.

The order of pragma statements matters.

The rootcint call looks like the following:
rootcint -f eventdict.cc -c -I. MyEvent.h MyJet.h LinkDef.h
Here eventdict.cc is the name of the dictionary file rootcint generates. -I. sets the include path, then the class headers are listed and finally LinkDef.h. This command can be added e.g. in a Makefile.

The library is then compiled from the dictionary code and the code describing the class. The library can then be loaded in ROOT
  root [] .L MyEvent.so
or used as a normal C++ library and linked with the analysis program.

A second way to add a class is with ACLiC:
  root [] gROOT->Macro("MyCode.C++")

Example: rootifying MyTrack class

In file MyTrack.h:

#include "TROOT.h"
#include "TLorentzVector.h"

Combining languages

```
class MyTrack : public
TLorentzVector {
   public:
      MyTrack();
...
   ClassDef(MyTrack,1) // macro
};
```

In file MyTrack.cc:

```
#include "MyTrack.h"
ClassImp(MyTrack) // macro
...
```

In file LinkDef.h

```
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;

#pragma link C++ class
MyTrack+;

#endif
```

In Makefile

```
eventdict.cc: MyTrack.h
        rootcint -f eventdict.cc -c -I. \
        MyTrack.h LinkDef.h
libMyTrack.so: $(OBJS)
        $(CXX) -shared -O *.o -o
libMyTrack.so
```

Python and C++

Here is a simple example of making a C++ module for Python. To support extension modules, Python API defines a set of functions, macros and variables that provide access to most aspects of the Python run-time system. The

Combining languages

Python API is incorporated in the C/C++ source code by including the header Python.h. To use the Python/C API, you need to have python-dev installed.

File testmodule.cc:

```
#include <Python.h>
#include <iostream>
static PyObject* testSrc(PyObject* self,
PyObject* args) {
    const char* name;
    if (!PyArg_ParseTuple(args, "s", &name))
return NULL;
    std::cout << name << std::endl;
    Py_RETURN_NONE;
}

static PyMethodDef MyMethods[] = {
    {"my_test", testSrc, METH_VARARGS,
"Some text."},
    {NULL, NULL, 0, NULL} # sentinel
indicating the end

};
```

```
static struct PyModuleDef MyModule = {
  PyModuleDef_HEAD_INIT,
  "MyModule","",-1,
  MyMethods
};

PyMODINIT_FUNC PyInit_MyModule(){
  return PyModule_Create(&MyModule);

}
```

The *self* argument is a straightforward translation from the Python argument list. It points to the module object for module-level functions, for a method it points to the object instance. The *args* argument is a pointer to a Python tuple object containing the arguments. The function PyArg_ParseTuple() checks the argument types and converts them to C (C++) values.

Combining languages

The return type must be understood by Python, and it must be a Python object. If the module is supposed to return a value, use function Py_BuildValue to build the return value, which is something like an inverse of the function PyArg_ParseTuple(). If the module is not returning a value, the corresponding Python function must return None. There are different ways to do this:

```
Py_RETURN_NONE; // or
Py_INCREF(Py_None);
return Py_None;
```

In addition to the actual module, Python needs a function for the module initialization. The initialization function must be named as "init"+module name.

The next step is to compile the module. For compiling, we need a script setup.py:

```
from distutils.core import setup, Extension

module1 = Extension('myModule', sources =
['testmodule.cc'])
setup (name = 'PackageName',
    version = '1.0',
    description = 'This is a demo package',

    ext_modules = [module1])
```

Compile:

```
python setup.py build
```

Install:

```
python setup.py install

–home=$HOME/python
```

Usage: write a short Python script to test the module. The new

Combining languages

module must be found in the Python path, here an installation in the $HOME directory is used.

In file test.py:

```
#!/usr/bin/env python
import sys,os

home = os.environ['HOME']
mypythonpath =
os.path.join(home,"python/lib/python")
sys.path.append(mypythonpath)

import myModule

myModule.my_test("Hello World")
```

Useful links

http://en.wikibooks.org/wiki/Python_Programming/Extending_with_C

http://docs.python.org/extending/extending.html

Python and ROOT, PyROOT

PyROOT is a Python extension module that allows the user to interact with any ROOT class from the Python interpreter. PyROOT provides Python bindings for ROOT: it enables cross-calls from ROOT/CINT into Python and vice versa, the intermingling of the two interpreters, and the transport of user-level objects from one interpreter to the other. PyROOT enables access from ROOT to any application or library that itself has Python bindings, and it makes all ROOT functionality directly available from the Python interpreter.

Combining languages

Useful links

http://root.cern.ch/drupal/content/how-use-use-python-pyroot-interpreter

http://wlav.web.cern.ch/wlav/pyroot/

To work with PyROOT, the env.var PYTHONPATH needs to be set in addition to the standard ROOTSYS.
 setenv PYTHONPATH ${ROOTSYS}/lib

PyROOT is available in Python via importing the top level Python module ROOT.py
 import ROOT

As a rule of thumb all "::"'s in Cint must be replaced with a dot "." in PyROOT.

It is also possible to import specific modules

from ROOT import TCanvas # or even
from ROOT import *
Cint: canvas = new TCanvas("canvas","",500,500);

Python: canvas = ROOT.TCanvas("canvas","",500,500)

To make the life easier, there are a number of working examples available in

$ROOTSYS/tutorials/pyroot

Example 1, plotting a TGraph. The function TGraph takes the number of elements, x-array of floats and y-array o f floats as input. The arrays can be provided by module array.

In file graph.py:

Combining languages

```
#!/usr/bin/env python

import ROOT
from array import array
ROOT.gROOT.SetBatch(True)

def main():
    x = array("d")
    y = array("d")
    x.append(1)
    y.append(1)
    x.append(3)
    y.append(2)
    n = len(x)

    canvas =
ROOT.TCanvas("someName","",500,500)
    canvas.SetFillColor(0)
    canvas.cd()

    frame =
ROOT.TH2F("frame","",2,0,4,2,0,3)
    frame.SetStats(0)
    frame.GetXaxis().SetTitle("x")
    frame.GetYaxis().SetTitle("y")
    frame.Draw()

    graph = ROOT.TGraph(n,x,y)
    graph.SetMarkerStyle(2)
    graph.SetMarkerSize(2)
    graph.SetLineColor(2)
    graph.Draw("PL")

    canvas.Print("graph.eps")
```

Here making the plot on screen is disabled with the line SetBatch(True). If the data is kept in arrays x = [], it must be converted to use array() before used as a TGraph argument.

Example 2, picking events from a tree. In physics analysis it is sometimes handy to be able to select a subset of events for a closer look. This example picks events based on a run number, luminosity block and an event number, three variables which allow a unique identification of an event in the CMS data. The events are listed in a txt file in a format run:lumi:event. The output file contains a tree with only the picked events included.

Combining languages

In file pickEvents.py:

```
#!/usr/bin/env python

import sys,os,re
import ROOT

root_re =
re.compile("(?P<rootfile>([^/]*))\.root$")
event_re =
re.compile("(?P<run>(\d+)):(?P<lumi>(\d+)):
(?P<event>(\d+))")

def main():
 if len(sys.argv) == 1:
   print
   print "# Usage:"+sys.argv[0]+" <root file>
   -pick <pick events file>"
   print
   sys.exit()

 rootfiles = []
 pickeventsfile = ""

 iarg = 1
 while iarg < len(sys.argv):
   if sys.argv[iarg]=="-pick" and
iarg<len(sys.argv)-1 :
      pickeventsfile = sys.argv[iarg+1]
      iarg += 1
   match = root_re.search(sys.argv[iarg])

   if match:
      rootfiles.append(sys.argv[iarg])
   iarg += 1
 events = getEvents(pickeventsfile)

 for file in rootfiles:
      pick(file,events)

def getEvents(filename):
   events = []
   fIN = open(filename,'r')
   for line in fIN:
      events.append(line.replace("\n",""))
   return events

def pick(filename,events):
   fIN = ROOT.TFile.Open(filename)
   fName = "picked.root"
   match = root_re.search(filename)
   if match:
      namebody = match.group("rootfile")

fName=filename.replace(namebody,"picked_"+namebo
   fOUT =
ROOT.TFile.Open(fName,'RECREATE')
   intree = fIN.Get("TTEffTree")
   tree = intree.CloneTree(0)

   for event in events:
      match = event_re.search(event)
      if match:

         selection ="run == " +

match.group("run")
```

Combining languages

```
selection+="&&lumi=="+match.group("lumi")
          selec-
tion+="&&event=="+match.group("event")
          picktree =
intree.CopyTree(selection)
          treelist = ROOT.TList()
          treelist.Add(picktree)
          tree.Merge(treelist)
    print "Saving",tree.GetEntries(),"events"
    tree.AutoSave()
    fIN.Close()
    fOUT.Close()
 if __name__ == "__main__":
    main()
```

For each event in the pick events text file, the tree passing the selection contains only that particular event, which is merged to an empty clone of the initial tree. No assumption about the tree contents needs to be made, except that it contains the fields used for the filtering.