Introducción a ROOT (TTree)

Curso de Técnicas Experimentales Avanzadas en Física Nuclear

Master Inter-universitario de Física Nuclear, curso 2017-2018

A. Tolosa Delgado

Instituto de Física Corpuscular (CSIC-Universitat de València)

Índice

Introducción

-El objetivo de es aprender a manejar la clase TTree.

-Empezaremos a usando TTree como una forma de entrada-salida, con los ejemplos que están colgados en la página de la asignatura.

-Veremos algunos de los métodos de la clase TTree

-Terminaremos comentando el ejercicio para casa

-Podeis preguntar y pararnos en cualquier momento.

-Documentación oficial:

https://root.cern.ch/doc/master/classTTree.html

-**RECOMENDACIÓN GENERAL**: compilar los códigos. Dentro de ROOT,

root -t

root[0] .L myScript.cxx++

¿Qué es un TTree?

-Como primera aproximación, podemos pensar en el TTree como una tabla:

Entry 0

BranchA			BranchB
Leaf1 (double)	Leaf2 (int)	Leaf3 (char)	Leaf 4(std::vector<int>)
5.24	0	‘s’	{0,1,2}
...	...	...	...

-En general, es una forma de guardar gran cantidad de objetos de la misma clase (no sólo POD).

-Por defecto, los datos son comprimidos (coste de tiempo)

-Gran cantidad de métodos (funciones) para su manejo

-**SIEMPRE** ligados a un TFile (fichero de ROOT)

Creando el primer TTree

-Cada variable se correspondería con una hoja

-Las hojas (leaf) se agruparían en ramas (branch)

-Podemos pensar que en cada rama hay una clase. Esta clase tendría como variables cada una de las hojas (con el mismo nombre y el tipo!)

-Si dos variables están relacionadas, es más eficiente ponerlas en la misma rama.

Creando el primer TTree (case A)

-Para definir una rama como conjunto de hojas se utilizan las siguientes letras clave:

C : a character string terminated by the 0 character
B : an 8 bit signed integer (Char_t)
b : an 8 bit unsigned integer (UChar_t)
S : a 16 bit signed integer (Short_t)
s : a 16 bit unsigned integer (UShort_t)
I : a 32 bit signed integer (Int_t)
i : a 32 bit unsigned integer (UInt_t)
F : a 32 bit floating point (Float_t)
D : a 64 bit floating point (Double_t)
L : a 64 bit signed integer (Long64_t)
l : a 64 bit unsigned integer (ULong64_t)
O : [the letter o, not a zero] a boolean (Bool_t)

Creando el primer TTree (case A & E)

```
class class4BranchA {
      double Leaf1={0.0}; // ordenar los miembros por tamaño
      int        Leaf2={ 0 };  // problemas de padding!!
      char      Leaf3={ '' };
};

void createFirstTree(){
TFile * ofile = TFile::Open( "firstTree.root" , "recreate" );
TTree * otree = new TTree ( "nameTree" , "titleTree" );
class4BranchA   myObj4BranchA;
std::vector<int> myObj4BranchB;

otree->Branch ( "BranchA" , &myObj4BranchA ,"Leaf1/D:Leaf2/I:Leaf3/B");
otree->Branch  ( "BranchB" , "std::vector<int>" , &myObj4BranchB);

ofile->Write();
ofile->Close();
}
```

Ver ejemplo 1: basic_atd.C

Acceso al TTree. Print, Show, Scan, StartViewer...

- TTree::Print() mostrará la estructura interna del TTree
- TTree::Show(n) mostrará la entrada n-ésima del TTree
- TTree::Scan("l1:l2:...","conditions") Similar a Show, pero se elige qué hojas mostrar; se pueden añadir condiciones
- TTree::StartViewer() Similar a TBrowser, pero con herramientas para proyectar un TTree
- TTree::MakeClass("myTreeReader"). ROOT revisa qué variables hay en el TTree y genera el esqueto de un script de análisis. Muy útil en los primeros pasos.

Acceso al TTree. Draw

Long64_t TTree::Draw (const char* varexp, const char* selection, Option_t* option = "")

- Draw permite proyectar una o varias hojas a histogramas

  -Las operaciones matemáticas están permitidas

- Se pueden usar condiciones (sobre la hoja que se está dibujando o cualquiera de las otras). Operaciones lógicas de C++ y cortes gráficos (TCutG) son aceptados.
- Se pueden especificar opciones de dibujo
- A veces es interesante redireccionar el histograma, y especificar manualmente sus parámetros

Acceso al TTree (ROOT6). Clase TTreeReader

En ROOT6 se utiliza la clase TTreeReader, con la siguiente sintaxis:

```
TFile * ifile = TFile::Open("basic.root","read");

TTree * inputTree = 0;

ifile->GetObject("aTree", inputTree);// Get() está obsoleto

if( inputTree == 0 ) return -2;
```

TTreeReader aReader (“areader” , ifile);

TTreeReaderValue < myEvent_t > anEntry(aReader, “myBranch.");

while (aReader.Next()) {

    std::cout << "x: " << anEntry->x << std::endl;

}

Acceso al TTree (ROOT5). TTree::SetBranchAddress()

En ROOT5 se utiliza la siguiente sintaxis:

```
TFile * ifile = TFile::Open("basic.root","read");
TTree * inputTree = 0;
ifile->GetObject("aTree", inputTree);// Get() está obsoleto
if( inputTree == 0 ) return -2;
myEvent_t * anEvent = new myEvent_t;
inputTree->SetBranchAddress("myBranch.", &anEvent );
Int_t nEntries( inputTree->GetEntries() );
for(Int_t iEntry = 0; iEntry< nEntries ; ++iEntry )  {
    inputTree->GetEntry( iEntry );
    std::cout << "x: " << anEvent->x << std::endl;
}
```

Apéndice. Diccionarios

Si queremos usar clases externas a ROOT, hay que definir los parámetros de IO que se necesitan -> “diccionario”

Hay tres formas de definir un diccionario

1) Cada vez que compilamos dentro de ROOT

```
root[0] .L myScript.cxx++;
```

2) Con la macro GenerateDictionary

```
gInterpreter->GenerateDictionary("myclass","myheader.h");
```

3) Con ROOTCINT (ver apéndice sobre compilación externa)

Véase

-→Detalles: https://root.cern.ch/faq/how-generate-dictionary

-→ LinkDef: https://root.cern.ch/selecting-dictionary-entries-linkdefh

Apéndice. Compilacion externa (Linux)

Como hemos visto, para compilar un programa de C++,

g++ main.cpp -o myExe

¿Y si queremos incluir clases de ROOT? **Hay que decir al compilador dónde están las librerías de ROOT**

g++ main.cpp `root-config --cflags --glibs` -o myExe

NOTA: en un script de ROOT **no** hay que definir una función **int main()**

Véase https://root.cern.ch/faq/how-generate-dictionary

Apéndice. Compilacion externa (Linux). Diccionario

Podemos generar un diccionario usando ROOTCINT.

Para ello, recordamos que la declaración de la clase debe estar en un fichero .hpp, y la definición de sus métodos en un fichero .cpp ó .cxx.

aClassTest.hpp

```
#ifndef __aClassTest_hpp__
#define __aClassTest_hpp__

class aClassTest
{
public:
  aClassTest();
  ~aClassTest();
  void Set( double i );
private:
  double Leaf1={-3.2};
};

#pragma link C++ class aClassTest+;

#endif
```

aClassTest.cpp

```
#ifndef __aClassTest_cpp__
#define __aClassTest_cpp__

#include "aClassTest.hpp"

aClassTest::aClassTest ( ) : Leaf1 ( 0 ) { }

aClassTest::~aClassTest ( ) { }

void aClassTest::Set(double i)
{
  Leaf1 = i;
  return;
}


#endif
```

Apéndice. Compilacion externa (Linux). Diccionario

Podemos generar un diccionario usando ROOTCINT.

Para ello, recordamos que la declaración de la clase debe estar en un fichero .hpp, y la definición de sus métodos en un fichero .cpp ó .cxx.

main.cpp

```
#include "TFile.h"
#include "TTree.h"
#include "aClassTest.cpp"

int main() {
  TFile * ofile = TFile::Open("test_c.root","recreate");
  TTree * otree = new TTree( "aTree","");

  aClassTest dummy;
  otree->Branch("br.", &dummy );

  for( int i = 0; i < 10; i+=2 )  {    dummy.Set( i );     otree->Fill();   }

  otree→Write();   ofile->Close();
  return 0; }
```

Apéndice. Compilacion externa (Linux). Diccionario

Podemos generar un diccionario usando ROOTCINT. La sintaxis es:

rootcint -f myDictionary.cxx -c aClassTest.hpp

El fichero myDictionary.cxx declara la clase con miembros extra necesarios para el IO de ROOT.

Posteriormente se añade como un fichero más en la compilación

g++ myDictionary.cxx main.cpp `root-config --cflags –glibs` -o myExe

Estas instrucciones pueden agruparse en un “Makefile”:

```
aClassTest: main.cpp aClassTest.hpp aClassTest.cpp
	@echo Makefile to compile in Linux
	@rootcint -f myDictionary.cxx -c aClassTest.hpp
	@g++ myDictionary.cxx main.cpp `root-config --cflags --glibs` -o aClassTest
```