21. November 2012

Root, C++ and User Defined Classes

a How-To

This manual gives a short overview of how to implement a user defined class into the root framework.

Introduction

Why and when should one be facing this problem? There are several cases, some may be:
You wrote your classes, for example an 'Event' class defining a certain type of observable of your experiment, using your own framework and you

- want to exchange your data with others
- use the *.root* container file to store your data

Working with any IDE (like Visual Studio, KDevelop...) you can link its compiler against the root libraries to write C++ code using all the root features. This allows as well to create your own C++ classes. To have them behave 'root-like' you would make them inherit from `TObject` implementing all its methods. This especially is necessary if you want to use the `->Write()` method to store objects in a *.root* container file. One may have noticed that there is a method called `->WriteObjectAny(...)`, but unlike its title it cannot store "any" object due to the fact, that this object for example needs a certain type of streamer method to be packed into *.root*.

In the root manual, chapter 'Adding a class', you may or may not find helpful information concerning this topic.

Overview

In general there are two steps to follow:

- adding necessary methods to the class [rootcint]
- compiling to a library class [compiler]

] both: [ACLiC]

At first you have to make your class 'root-like' adding all the necessary methods to be a `TObject`. This is what rootcint does. Afterwards you have to use the compiler of your system to create a library, which can then be implemented into your IDE. Both of these steps can be done using ACLiC, which is also provided by the root framework.

The User Defined Class

Information on how to implement a user defined class in general may be given in any textbook. A complete example of an 'Event' template class you can also find in `(rootsys)\test\Event.h` and `(rootsys)\test\Event.cxx`.
Your user defined class should consist of a header file (*.h*), containing the variable and method declarations as well as the `#includes`, and a body (e.g. *.cpp*) containing the code and `#including` the *.h* file. Your class has to have at least a default constructor ().

Preparation:

- **header**: declare your class as a `TObject` like `class [classname]:public TObject` (you have to `#include TObject.h`)
- **header**: add `ClassDef([classname],1);` at the end of the file just before the closing `};` statement. The number represents the version of the class.
- **body**: add the following statement

```
#ifndef __CINT__
ClassImp([classname]);
#endif
```

Caution: Don't use std:vector<> as a parameter or return value in any methods of your class. There are some unexplained memory issues.

The rootcint + compiler method

As I would rather recommend to use ACLiC, I will give only a short overview.

- In order to generate a file, which can be compiled, rootcint needs some information, which you have to put into a file named `LinkDef.h`. An example is located at `(rootsys)\test\EventLinkDef.h` and you can find an endless description of the possible `#pragma` statements in the root manual.
  Finally you call rootcint with the LinkDef as the last parameter:
  `rootcint -f [classname]dict.cxx -c [all header files] LinkDef.h`
  This generates a file `[classname]dict.cxx` containing all methods to be a root class.

- Compile it! This step is different for different operating systems:
  - **Linux**: use for example `gmake`. This generates a library, called Shared Object (*.so*). This file you just have to load into root using `.L [filename]`
  - **Windows**: Making use of an IDE you have to start a project to get a library file (.lib) at the end. You have to set up all the environment variables to link against root and then compile it. Then you have to simply add the library file to the collection of root libraries you already added to your original project.

The ACLiC method

ACLiC is implemented in the root environment, it can be called adding a `+` or a `++` to the `.L` command. Simply call:

```
.L (classbodyfile)++
```

and it will generate a library file, which you then have to load into your environment (like you already did when using an IDE to write your classes).

You could also implement it in a more straight forward way:

```
if (!TClass::GetDict("[classname]")) {
gROOT->ProcessLine(".L [classbodyfile].cpp++");}
```