Adding a ROOT Class and linking with ROOT using GNUMake

Short Guide

David Gallacher

Carleton University

June 2020

Abstract

While going through the exercise of adding and linking a series of custom classes with ROOT libraries and a program, I felt the lack of a guide aimed at a beginner/intermediate level (Such as myself). To remedy this I've written this guide, which will go through defining a simple ROOT Class that inherits from TObject, the necessary intermediate steps and making a GNUMake file to compile and link the class with ROOT libraries.

The code for this project is available on Github here: https://github.com/dgallacher1/simpleclass

1 Defining a simple class

There are several ways to write compiled code using ROOT, in this guide we have a simple executable program that calls a custom class that has been linked with ROOT through a shared library with the compilation and implementation done using a Makefile. A more detailed guide for adding a class to ROOT is here: https://root.cern.ch/root/htmldoc/guides/users-guide/AddingaClass.html.

To start with we want to define a simple ROOT/C++ class that inherits from TObject so that we can take advantage of ROOTs I/O and inspection.

Below we have an example of a simple class that has a single private data member and simple Get/Set functions and a single function defined in the source file which returns the square of the value. This file is called "SimpleClass.hh"

```
// A simple class
```

```
#ifndef ROOT_SimpleClass
#define ROOT_SimpleClass

#include "TObject.h"

class SimpleClass : public TObject {
private:
  // Private data members
  Int_t       value;

public:
    //Constructor and destructor
   SimpleClass();
   ~SimpleClass();

   void        Init();
   void        Clear(Option_t *option = "");

   // Public member functions
    Int_t         GetValue(){return value;}
    void             SetValue(Int_t _value){value = _value;}

    Int_t         GetSquare();

   ClassDef(SimpleClass,1)  //Root Class version
};
#endif
```

In the source file "SimpleClass.cxx" we define the public member functions listed in the header.

```
#include "SimpleClass.hh"

ClassImp(SimpleClass)

SimpleClass::SimpleClass()
{
  Init(); // Initialize values
}
```

```
SimpleClass::~SimpleClass()
{
  Clear();// Clear our values, this allows us to clear arrays and
      call a sequential Clear() function for nested classes.
}


/// Reset class
void SimpleClass::Clear(Option_t *option)
{

}

///Initialize parameters
void SimpleClass::Init()
{
  value = 0.0;
}

Int_t SimpleClass::GetSquare()
{
  return value*value; // Returns the square of the value
}
```

Now that we have our simple class we want to compile with ROOT we can proceed to the next steps.

2 LinkDef Files and additional ingredients

Once we have a simple class defined that we are happy with and want to take advantage of we need a few more ingredients to complete our recipe.

- LinkDef.h file to tell ROOT what to include in our dictionary
- A dictionary that gives ROOT instructions on the functions within our class(s). This tells ROOT what's inside the library and how to read it.
- A shared library file that we link with the ROOT and other libraries that allows us to access our shiny new class and functions inside of a macro, function or other classes.

The LinkDef file tells the rootcint/rootcling interpreter what to include in the dictionary that will be used to create our shared library. It is mandatory and must be called either "LinkDef.h" or "linkdef.h". This is described well here: https://137.138.13.143/selecting-dictionary-entries-linkdefh.

Most LinkDef files have the same format. An example is given below.

```
#ifdef __CINT__

#pragma link off all globals;
#pragma link off all classes;
#pragma link off all functions;
#pragma link C++ nestedclasses;

#pragma link C++ class SimpleClass+;

#endif
```

The #pragma tells the compiler what it needs to generate dictionaries for, we turn off some of the defaults and add our simple class, adding a class requires the '+' symbol. More details are given in the link above.

Importantly the "LinkDef.h" must be linked **last** when creating the dictionary with rootcint(or rootcling). This is described with examples here: https://root.cern.ch/interacting-shared-libraries-rootcling .

3 Adding it all together with a Makefile

An easy way to avoid having to issue all the recompiling commands when you are making changes to a single part of your class/program is to utilize the power of the make program. A good introduction to make is given here: https://opensource.com/article/18/8/what-how-makefile but the full details can be found in the GNUMake manual here https://www.gnu.org/software/make/manual/. Using a makefile simplifies compilation by issuing all the required commands automatically and will also determine which files need to be updated based on when the files were last accessed. This will save time especially when developing medium/large projects with many files and components.

```
#Root libraries
ROOTLIBS      := $(shell $(ROOTSYS)/bin/root-config --libs)
ROOTGLIBS     := $(shell $(ROOTSYS)/bin/root-config --glibs)

#Variables and options
```

```
OPT             := -O2
#Compliler of choice
CXX             := g++
CXXFLAGS        := -Wall -pthread $(OPT) -fPIC -I$(ROOTSYS)/include
LD              := g++
LDFLAGS         := $(OPT) -pthread
LIBS              := $(ROOTGLIBS) -L/usr/X11R6/lib -lX11 -lXpm


#include all header files for linking here
HDRS := SimpleClass.hh

#Objects files for each class
#This will automatically create an object file for each header
    file "Header.hh" called "Header.o" in the list "HDRS"
OBJS := $(HDRS:.hh=.o)
#Source files for each class, here we also create an array of the
    source files corresponding to the headers
#This helps us reduce the number of places to input variables and
    reduce the chance of typos causing issues
SRCS := $(HDRS:.hh=.cxx)

#Since we dont create a file called "all" or "clean" with our
    rules, we declare them as phony targets
#A good description for this is here : https://www.gnu.org/
    software/make/manual/html_node/Phony-Targets.html
.PHONY: all clean

#Having a function called "all" ensures that all of our rules will
     be ran when we call "make"
#otherwise only the first rule will be called
#Call "make clean" to remove clean the directory and start
    compilation fresh.
all: lib Dictionary.C program

lib: libtree.so

#Here we create a shared library to use in our simple program
```

```
#We're linking our dictionary with all of our object files.
libtree.so: $(OBJS) Dictionary.o
        @echo "Creating shared library.."
        $(LD) $(LDFLAGS) -shared -o $@ $^ $(LIBS)

#To illustrate what's going on below we compile the dictionary
    file independently
#The
Dictionary.o: Dictionary.C
        @echo "Creating dictionary object.."
        $(CXX) -c $(CXXFLAGS) $<

Dictionary.C: $(HDRS) LinkDef.h
        @echo "Creating dictionary.."
        $(ROOTSYS)/bin/rootcint -f $@ -c $(CXXFLAGS) -p $^

#This will be a simple program we use to test our class
    implementation
program: program.o
        @echo "Creating program.."
        $(LD) $(LDFLAGS) -o $@ $< $(LIBS) -L/path/to/library/ -
            ltree

#This rule helps us build our individual machine-readable object
    files for header files listed in HDRS
%.o: %.cxx
        $(CXX) -c $(CXXFLAGS) $<

clean:
        @echo "Cleaning up..."
        rm -f *.o
        rm -f *~
        rm -f *.so
        rm -f Dictionary.*
        rm -f program
```

This Makefile can serve as a template for a project and can be expanded upon. We make use of a few variables defined at the top of the file, this reduces the number of places we need to edit in the Makefile and is very helpful if we want to change our compiler settings (through

flags) or library inclusions. We also make use of a few dynamic variables to simplify our code, at the cost of some readability. These are outlined below but for a detailed description of how to use variables in GNUMake refer to the user manual.

Two things to note, there are different types of variable definitions in make, simple expanded variables defined with ":=" which work very much like normal variables in C++ and are defined once at declaration, and recursively expanded variables set with "=" which are defined whenever these variables are called (See Section 6.2 of the GNUMake manual for details) which can lead to infinite loops. The second thing to keep in mind is that Makefiles are whitespace sensitive, and spaces and tabs are interpreted differently.

- $@ - Gets the literal string of the target file name for the rule.
- $∧ - Gets the names of all prerequisites of the rule
- $< - Gets the name of the first prerequisite
- Calling @ before a shell command silences the command output
- A variable is accessed by $() or ${}

For a more detailed description see GNU manual here: https://www.gnu.org/software/make/manual/html_node/Automatic-Variables.html

In this Makefile we do a number of important steps:

- Create a machine-readable object file for our source file defined in SimpleClass.hh/cxx
- Create a dictionary for our class implementation
- Create an object file for our dictionary
- We link all of our object files (2 in this case) into a shared library (libtree.so) for use in our test program
- We created an executable and compiled our program linked with the ROOT libraries.

In order to have our shared library available at run-time for our "program" we need to make our library available. We do this by adding the library path to our "LD_LIBRARY_PATH", this is done by calling
"export LD_LIBRARY_PATH=/home/username/path/to/library/:$LD_LIBRARY_PATH"
We can do this in terminal or add it as a command to our bash profile so it adds automatically at login.

This is described well in detail here:
https://www.cprogramming.com/tutorial/shared-libraries-linux-gcc.html.

Now that we've done this we should have access to the public member functions of "SimpleClass" in our test program as well as ROOT library functions.

We will also be able to access our library from the ROOT console as well. Open ROOT by calling "root" in the command line. Once we are inside the interactive terminal we can input the following command:

```
//Load the shared library into the interpreter

gSystem->Load("path/to/library/libtree.so");

//Now we can access our class.
SimpleClass *class = new SimpleClass();
```

By loading the shared library in the interactive ROOT console we can also view our class variables in a TBrowser if they are saved as part of a TFile in a TTree.

4 Simple program

In the previous section we also created a simple executable program "program". This is defined by "program.cxx" here.

```
//Here we have a simple program that uses our custom ROOT class.

#include "SimpleClass.hh"

#include "TROOT.h"
#include "TObject.h"

#include <iostream>

using namespace std;

int main(int argc, char **argv){

    Int_t val;
    if(argc==2){
      val = atoi(argv[1]);
    } else {
      cout <<"./program value"<< endl;
      return 0;
```

```
    }

    SimpleClass *simpleclass = new SimpleClass();

    simpleclass->SetValue(val);

    Double_t valsq = simpleclass->GetSquare();

    cout << "Value squared = " << valsq << endl;

    delete simpleclass;
    simpleclass =0;

    return 0;
}
```

This program can be expanded upon. If for example we wanted to add our class into a ROOT TTree we would include something like below.

```
#include "TFile.h"
#include "TTree.h"
#include "TBranch.h"
.
.
.
//A file to save our tree to.
TFile *fileout = new TFile("treefile.root","RECREATE");
TTree *tree = new TTree("tree","A tree with a simple class");

SimpleClass *simpleclass = new SimpleClass();
tree->Branch("classbranch",&simpleclass,8000,1); // Create a
    branch for our class with a buffer size of 8000 and 1:1
    splitting

//Set the value
simpleclass->SetValue(100);
// Fill the tree
tree->Fill();
//Clear the class and reinitialize for the next fill loop.
simpleclass->Clear();
```

```
simpleclass->Init();
.
.
fileout->cd(); // Set the fileout to be the current directory
tree->Write("tree"); // Write our tree to the file
.
.
```

5 Conclusion

I hope this helps beginners get started with using ROOT in a more comprehensive manner and avoids some of the inevitable headaches from chasing down information in forgotten ROOT forums chased by the ghost of Rene Brun.