Introduction to ROOT

Prof. Silvia Masciocchi
dr. Federica Sozzi

Heidelberg University

Day 5 – Projects with ROOT

Based on the slides created by dr. Jens Wiechula, Frankfurt University

Outline

- Projects with ROOT
  - Adding own classes to ROOT
  - Creating a software library
  - Makefiles
  - Using own software libraries in ROOT
- Few more useful information about ROOT

Adding own classes to ROOT

Disclaimer

- The way of writing code and the layout of a project is subject to personal taste
- A working example will be shown, but perhaps there are better ways
- Many things will only be treated at the very surface
- Be advised to carefully read the ROOT manual where much more information is given
- Many helpful things are also given in the ROOT tutorials

Adding own classes to ROOT

Introduction

- ROOT offers simple methods to add own classes to ROOT and profit from the additional functionalities
  - Writing classes to file
  - Adding the class to a ROOT collection
  - Inclusion in the ROOT prompt
- This requires to add definitions in the own code
  - Inheritance from TObject
  - The 'ClassDef' in the header
  - The 'ClassImp' in the implementation (needed for the automatic documentation)
  - The implementation of a default constructor

Adding own classes to ROOT

example of header

Inheritance from TObject

```
class myEvent : public TObject {
public:
  myEvent();
  myEvent(const myEvent &ev);
  virtual ~myEvent();

  void SetVx(Double_t vx) { fVx = vx; }
  Double_t GetVx() const { return fVx; }

private:
  Double_t fVx;                    // x vertex of the event

  ClassDef(myEvent,1);            // event information
};
```

code/myEvent.h

Usage of macro ClassDef

Adding own classes to ROOT

Inheritance from TObject

```
#include "myEvent.h"

ClassImp(myEvent)

myEvent::myEvent()
: TObject()
, fVx(0.)
, fParticles("myParticle")
{
  // default constructor
}

//_____________________________________________
_____________
myEvent::myEvent(const myEvent &ev)
: TObject(ev)
, fVx(ev.fVx)
, fParticles(ev.fParticles)
{
  // copy constructor
}
```

Note on inheritance:

When the constructor of the derived class (myEvent) is called, the default constructor of the base class (TObject) is called

- One can specify the call to the base constructor explicitly

code/myEvent.cxx

Adding own classes to ROOT

ClassDef

- The ClassDef macro has the following form:

```
ClassDef(ClassName,ClassVersionID); //The class title
```

ClassName	Name of the class (what is defined behind 'class')
ClassVersionID	The 'layout version' of the class. NOTE: if ClassVersionID is '0' no automatic I/O capabilities will be generated for this class! If I/O is wished ClassVersionID needs to be >=1. If the class is first written use '1'. The ClassVersionID must be increased if data members are added or removed. This will assure backward compatibility and allows ROOT to read/write objects even if the layout of the class was changed. ROOT writes the 'streamer version' also to file.
//The class title	This sets the title of the class displayed in ROOT and the ROOT documentation

Adding own classes to ROOT

ClassImp

- The ClassImp macro has the following form:

```
ClassImp(ClassName)
```

Usually the macro is put before the first constructor in the implementation file

ClassName	Name of the class (what is defined behind 'class')

```
#include "myEvent.h"

ClassImp(myEvent)

myEvent::myEvent()
: TObject()
, fVx(0.)
, fParticles("myParticle")
{
  // default constructor
}
```

Usage of macro ClassImp

code/myEvent.cxx

Adding own classes to ROOT

The default constructor

- For I/O purposes ROOT requires the implementation of a default constructor
  - A constructor without arguments
  - A constructor which assigns default values to all parameters
- The default constructor will be used to create an instance of this object before calling the streamer and filling the data members

Adding own classes to ROOT

The default constructor

- Be sure that you do not allocate any space for embedded pointer objects in this constructor.
- This space will be lost (memory leak) while reading in the object.

```
class T49Event : public TObject {
private:
  Int_t fId;
  TCollection *fTracks;
...
public:
  // Error space for TList pointer will be lost
  T49Event() { fId = 0; fTrack = new TList; }
  // Correct default initialization of pointer
  T49Event() { fId = 0; fTrack = 0; }
...
};
```

https://root.cern.ch/root/htmldoc/guides/users-guide/ROOTUsersGuide.html#the-default-constructo

Adding own classes to ROOT

Streamer

But what does it mean “write objects to a file” ?

When we say, “writing an object to a file”, we actually mean writing the current values of the data members. The most common way to do this is to decompose (also called the serialization of) the object into its data members and write them to disk.

The decomposition is the job of the Streamer.

Every class with ambitions to be stored in a file has a Streamer that decomposes it and “streams” its members into a buffer

In ROOT the streamer can be automatically generated for any class that in its implementation follows the “rules” explained in the previous slides

https://root.cern.ch/root/htmldoc/guides/users-guide/InputOutput.html#streamers

Adding own classes to ROOT

rootcint/rootcling

- rootcint/rootcling creates a so called 'dictionary'
- This is a source file that once compiled, linked into a library or executable and loaded into a process will provide all the information needed about a type or variable

Generate dictionary

Header files for which we want a dictionary

```
ROOT5
rootcint eventdict.cxx -c myEvent.h myParticle.h

ROOT6
rootcling eventdict.cxx -c myEvent.h myParticle.h
```

Adding own classes to ROOT

rootcint/rootcling – example output

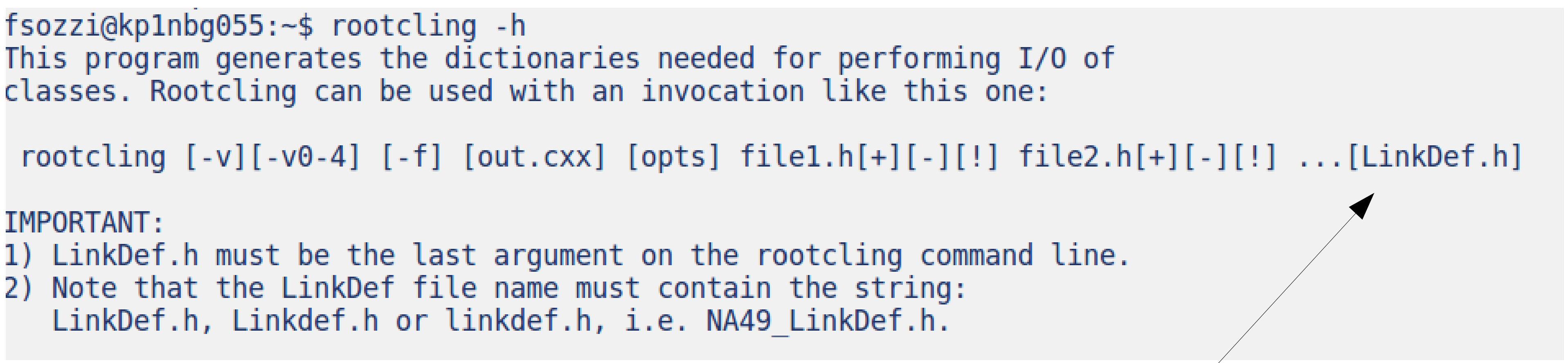
It contains the automatically generated

- Streamer() function for the relevant classes.
- ROOT 5: ShowMembers() function used to make a dump of the class' data members

```
//______________________________________________________________________
void myEvent::Streamer(TBuffer &R__b)
{
   // Stream an object of class myEvent.

   if (R__b.IsReading()) {
      R__b.ReadClassBuffer(myEvent::Class(),this);
   } else {
      R__b.WriteClassBuffer(myEvent::Class(),this);
   }
}

//______________________________________________________________________
void myEvent::ShowMembers(TMemberInspector &R__insp)
{
      // Inspect the data members of an object of class myEvent.
      TClass *R__cl = ::myEvent::IsA();
      if (R__cl || R__insp.IsA()) { }
      R__insp.Inspect(R__cl, R__insp.GetParent(), "fVx", &fVx);
      R__insp.Inspect(R__cl, R__insp.GetParent(), "fParticles", &fParticles);
      R__insp.InspectMember(fParticles, "fParticles.");
      TObject::ShowMembers(R__insp);
}
```

Adding own classes to ROOT

LinkDef.h file



```
fsozzi@kp1nbg055:~$ rootcling -h
This program generates the dictionaries needed for performing I/O of
classes. Rootcling can be used with an invocation like this one:

 rootcling [-v][-v0-4] [-f] [out.cxx] [opts] file1.h[+][-][!] file2.h[+][-][!] ...[LinkDef.h]

IMPORTANT:
1) LinkDef.h must be the last argument on the rootcling command line.
2) Note that the LinkDef file name must contain the string:
   LinkDef.h, Linkdef.h or linkdef.h, i.e. NA49_LinkDef.h.
```

In ROOT a special header file 'LinkDef.h' can be used to tell 'rootcint/rootcling' which classes should be added to the dictionary

Adding own classes to ROOT

LinkDef.h file

- The LinkDef file contains statements as shown below.

  For more details see the ROOT manual or https://root.cern.ch/selecting-dictionary-entries-linkdefh

```
#pragma link C++ class myEvent+;
#pragma link C++ class myParticle+;
```

- For each class that should be included in the ROOT functionality a '#pragma link C++ class <name>[type]' statement needs to be given

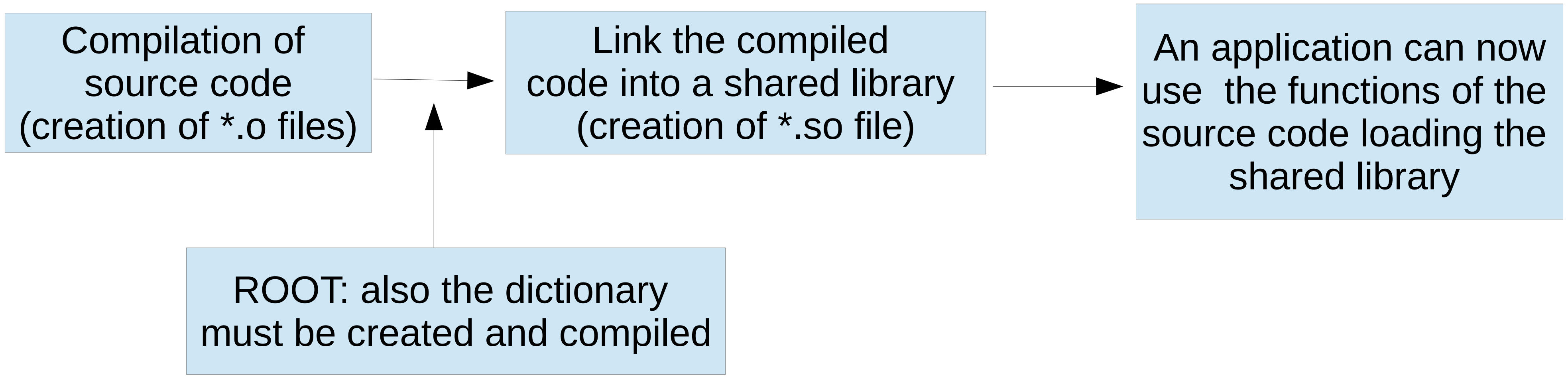
name	Name of the class (what is defined behind 'class')
type	Specifies which kind of additional functionality will be build
	+: All available
	-: No streamer
	!: No '>>' operator

MyPackageLinkDef.h

Creating a software library

Introduction

- Usually when developing own code one creates a software library which can be linked against
- A software library is created by 'linking' compiled code (the object files) into a shared library
- This process is done by the 'linker'
- Compilation and linking is usually done in the makefile, such that the user only needs to type 'make' in the shell


Compilation of source code (creation of *.o files)
Link the compiled code into a shared library (creation of *.so file)
An application can now use the functions of the source code loading the shared library
ROOT: also the dictionary must be created and compiled

Creating a software library

Introduction

- Compilation and linking are two different steps
- in the simple examples of the first day, “HelloWorld.cpp” and “classes.cpp”, we did not distinguish the two phases and let g++ do all for us
- For complex programs, it makes sense to distinguish the two steps. E.g. imagine that you modify only a source file: you will need only to recompile that file (and not all the source files) and link it. Or imagine that you want to create a shared library that will be used later from another application

http://www.cprogramming.com/compilingandlinking.html

Makefiles

Introduction

- 'Makefile' takes over all necessary steps of the compilation and linking
- The GNU make system is a powerful way of automatising the compilation of code and creating software libraries
- For the definition a steering file with the default name 'Makefile' is used
- It is a very complex system, no details will be discussed, only a working version...
- The documentation can be found here:

http://www.gnu.org/software/make/manual/make.html

Makefiles

Introduction 2

- What happens in Makefiles is steered by 'targets'
- Targets are identifiers followed by a ':'
- The first target in a Makefile will be what is done by default
- Targets can depend on each other
- The process is started by calling 'make <target>', if <target> is empty, the default target is used

Makefiles

An (already complex) example

```
PACKAGE = MyPackage

default-target: lib$(PACKAGE).so

#include the definitions for this package
include lib$(PACKAGE).pkg

# add the root dictionary to the list of sources
SRCS_package := $(SRCS) G__$(PACKAGE).cxx
# build the list of objects (compiled code)
OBJS_package := $(SRCS_package:.cxx=.o)

# link the objects to a software library
lib$(PACKAGE).so: $(OBJS_package)
	@echo "Linking" $@ ...
	@/bin/rm -f $@
	@$(LD) $(SOFLAGS) $(LDFLAGS) $^ -o $@
	@chmod a+x $@
	@echo "done"

# compile the code
%.o:    %.cxx %.h
	$(CXX) $(CXXFLAGS) -c $< -o $@

G__$(PACKAGE).cxx: $(HDRS) $(DHDR)
	@echo "Generating dictionary ..."
	rootcint -f $@ -c $(CINTFLAGS) $(INC) $^
```

Use cint for ROOT5
cling for ROOT6

Makefiles

The package definition

- For simple packages this Makefile can be re-used
- Only the code definition needs to be changed
- This is done in the 'lib<package>.pkg' file:

```
#-*- Mode: Makefile -*-

# list of source files to be compiled
# needs to be separated by a space
# or if a line break is used the line
# must end with a '\'
SRCS= code/myEvent.cxx \
      code/myParticle.cxx

HDRS= $(SRCS:.cxx=.h)

# This is name of the linkdef file of the package
DHDR= MyPackageLinkDef.h

# here one could define paths to header file needed
# to compile the package, separated by space
EINCLUDE:=-I./code
```

Creating a software library

An example

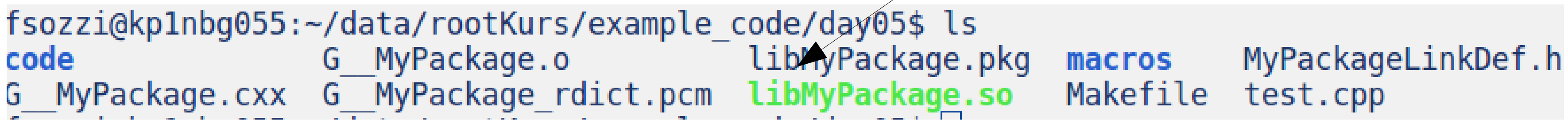
```
example_code/day05$ make

g++ -g -Wall -fPIC -pthread -m64 -I/data/Work/software/root/v5-34-02/include
-W -Wall -Weffc++ -Woverloaded-virtual -fPIC -pipe -fmessage-length=0
 -Wno-long-long -ansi -Dlinux -I. -I./code -g -c code/myEvent.cxx -o code/myEvent.o

g++ -g -Wall -fPIC -pthread -m64 -I/data/Work/software/root/v5-34-02/include -W
 -Wall -Weffc++ -Woverloaded-virtual -fPIC -pipe -fmessage-length=0
-Wno-long-long -ansi -Dlinux -I. -I./code -g -c code/myParticle.cxx -o code/myParticle.o

Generating dictionary ...
rootcint -f G__MyPackage.cxx -c  -I. -I./code code/myEvent.h
code/myParticle.h MyPackageLinkDef.h

g++ -g -Wall -fPIC -pthread -m64 -I/data/Work/software/root/v5-34-02/include
-W -Wall -Weffc++ -Woverloaded-virtual -fPIC -pipe -fmessage-length=0
-Wno-long-long -ansi -Dlinux -I. -I./code -g -c G__MyPackage.cxx -o G__MyPackage.o

Linking libMyPackage.so ...
c++ -shared -O2 -m64 code/myEvent.o code/myParticle.o G__MyPackage.o -o libMyPackage.so
done
```

Compile myEvent.cxx, myParticle.cxx and create objects

Generate ROOT dictionary

Compile dictionary, create object

Link the 3 objects and create the .so



```
fsozzi@kp1nbg055:~/data/rootKurs/example_code/day05$ ls
code               G__MyPackage.o          libMyPackage.pkg  macros    MyPackageLinkDef.h
G__MyPackage.cxx   G__MyPackage_rdict.pcm  libMyPackage.so   Makefile  test.cpp
```

Using the library in ROOT

Including the library in CINT

- In CINT a library can be loaded with

```
root [0] .L libMyPackage.so
```

- The following line can be used in CINT or inside the code (macros)

```
root [1] gSystem->Load("libMyPackage.so")
```

→ Check that you can access the classes myEvent and myParticle from the ROOT prompt after this command

- If you use additional root classes in your library (like matrices, function, I/O, …), you have to load the corresponding libraries before loading your own library

  (also e.g. via gSystem->Load(...))

  See macros/loadLibs.C (and consider comments in code!)

Using the library in ROOT

Including the library before compilation

- If the library shall be used in CINT or ACLiC compilation, it needs to be loaded before
- In addition ACLiC needs to be told where to
  - Search for the include files

```
root [1] gSystem->AddIncludePath("-I/path/to/my/headers");
```

  - Search for libraries

```
# this is done in the shell by setting the following
# environment variable
export LD_LIBRARY_PATH=/path/to/my/lib:$LD_LIBRARY_PATH
```

LD_LIBRARY_PATH is not ROOT specific, but a Linux wide variable

Example

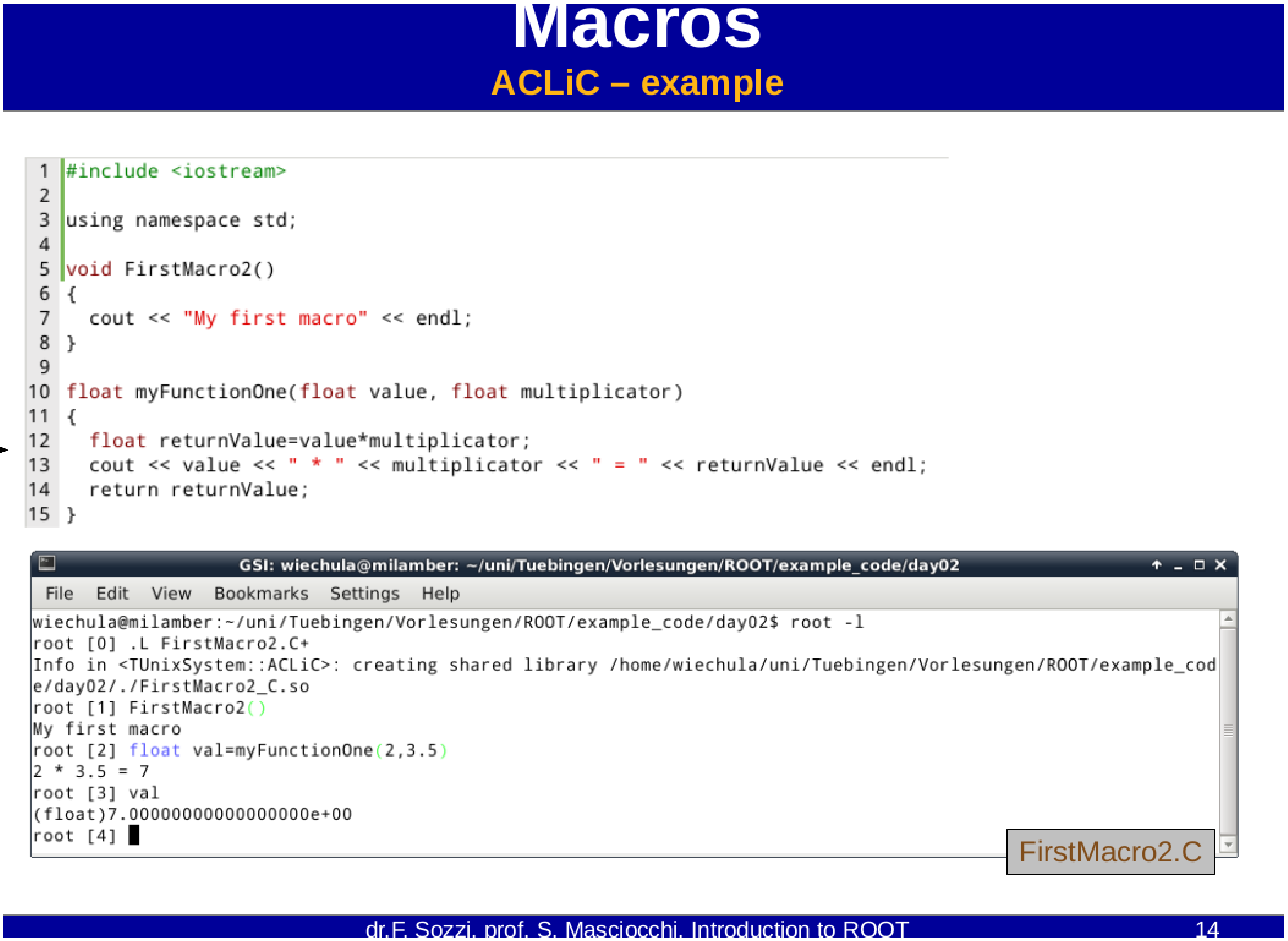
```
root[] .L libMyPackage.so
root[] gSystem->AddIncludePath("-I./code")
root[] .L createEvent.cpp++
```

Macro compilation in ACLIC

Actually we have already encountered shared libraries in ROOT for simple cases: remember the compilation of a macro inside CINT ?

Lecture 2 →

This operation compiles, creates the dictionary and generate the shared library in one go, without Makefile!



Macros

ACLiC – example

```cpp
#include <iostream>

using namespace std;

void FirstMacro2()
{
  cout << "My first macro" << endl;
}

float myFunctionOne(float value, float multiplicator)
{
  float returnValue=value*multiplicator;
  cout << value << " * " << multiplicator << " = " << returnValue << endl;
  return returnValue;
}
```

GSI: wiechula@milamber: ~/uni/Tuebingen/Vorlesungen/ROOT/example_code/day02
File Edit View Bookmarks Settings Help

```
wiechula@milamber:~/uni/Tuebingen/Vorlesungen/ROOT/example_code/day02$ root -l
root [0] .L FirstMacro2.C+
Info in <TUnixSystem::ACLiC>: creating shared library /home/wiechula/uni/Tuebingen/Vorlesungen/ROOT/example_cod
e/day02/./FirstMacro2_C.so
root [1] FirstMacro2()
My first macro
root [2] float val=myFunctionOne(2,3.5)
2 * 3.5 = 7
root [3] val
(float)7.00000000000000000e+00
root [4] █
```

FirstMacro2.C

dr.F. Sozzi, prof. S. Masciocchi, Introduction to ROOT 14

Adding own classes to ROOT

More on class data members

The streaming of data members can also be steered in the header file

```
class Event : public TObject {
private:
   TDirectory    *fTransient;            //! current directory
   Float_t     fPt;                   //! transient value
char           fType[20];
Int_t          fNtrack;
Int_t          fNseg;
Int_t          fNvertex;
UInt_t         fFlag;
Float_t        fTemperature;
EventHeader    fEvtHdr;           //|| don't split
TClonesArray  *fTracks;           //->
TH1F          *fH;                //->
Int_t          fMeasures[10];
Float_t        fMatrix[4][4];
Float_t       *fClosestDistance;  //[fNvertex]
...
```

https://root.cern.ch/root/htmldoc/guides/users-guide/InputOutput.html#streamers

Adding own classes to ROOT

More on class data members

The streaming of data members can also be steered in the header file

```
class Event : public TObject {
private:
   TDirectory    *fTransient;             //! current directory
   Float_t     fPt;                    //! transient value
char           fType[20];
Int_t          fNtrack;
Int_t          fNseg;
Int_t          fNvertex;
UInt_t         fFlag;
Float_t        fTemperature;
EventHeader    fEvtHdr;             //|| don't split
TClonesArray  *fTracks;             //->
TH1F          *fH;                  //->
Int_t          fMeasures[10];
Float_t        fMatrix[4][4];
Float_t       *fClosestDistance;   //[fNvertex]
...
```

If data members don't need to be written to file ('transient data members'), put a "//!" directly behind the member

https://root.cern.ch/root/htmldoc/guides/users-guide/InputOutput.html#streamers

Adding own classes to ROOT

More on class data members

The streaming of data members can also be steered in the header file

```
class Event : public TObject {
private:
   TDirectory    *fTransient;            //! current directory
   Float_t      fPt;                  //! transient value
char           fType[20];
Int_t          fNtrack;
Int_t          fNseg;
Int_t          fNvertex;
UInt_t         fFlag;
Float_t        fTemperature;
EventHeader    fEvtHdr;
TClonesArray  *fTracks;
TH1F          *fH;
Int_t          fMeasures[10];
Float_t        fMatrix[4][4];
Float_t       *fClosestDistance;  //[fNvertex]
...
```

size of the dynamic c-array

In case dynamic c-arrays are used in a class, the streamer needs to know about the size, this requires an additional data member of type 'Int_t' which stores the size of the array

pointer to the array

https://root.cern.ch/root/htmldoc/guides/users-guide/InputOutput.html#streamers

Small software projects

Layout

- Even for small projects it is advisable to structure the code
- E.g. simple structure with one subdirectory each for
  - The code defining the classes
  - Code written in macros

```
libMyPackage.pkg  Makefile  MyPackageLinkDef.h

code:
myEvent.cxx  myEvent.h  myParticle.cxx  myParticle.h

macros:
loadlibs.C
```

More useful information about ROOT

- Logon and logoff scripts, root history file
- Global variables
- More useful classes and name spaces

Logon and logoff scripts

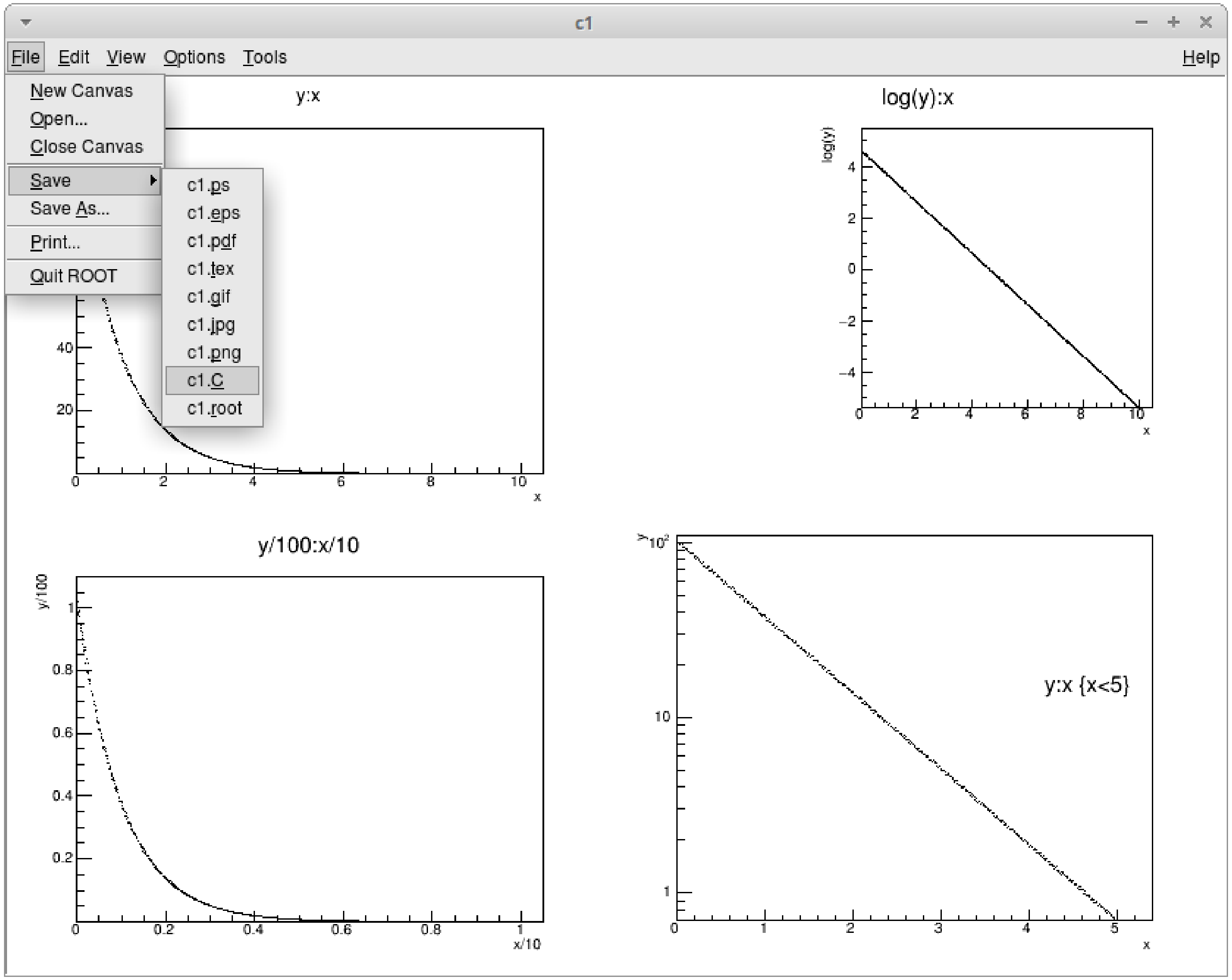
- When root is started and stopped, macros are automatically executed (unless the '-n' option is given)
- The name of the script is defined in the root configuration file '~/.rootrc'
- Per default they are '$(HOME)/rootlogon.C' and '$(HOME)/rootlogoff.C'
- They can e.g. be used
  - To automatically set include paths for ACLiC if external code is used
  - Define a global style (see below)

ROOT command history

- Every command typed in the ROOT prompt is saved in the file ~/.root_hist

  It can be useful to extract ROOT commands to use in your macros

How to pass from GUI to code



If you save a canvas as ".C" ,ROOT will create a macro reproducing the canvas

This is particularly useful when you modify something from the GUI and want to find quickly the corresponding command to do it

Global variables

Summary

- ROOT defines a few global variables that are available anywhere in the code:

  they all start with “g”
- gROOT (TROOT) session information, entry point to the ROOT system (*TROOT object is essentially a container of several lists pointing to the main ROOT objects*.)
- gStyle (TStyle) the current global graphics style
- gSystem (TSystem) the underlying OS (Linux, Windows)
- gFile (TFile) current open file
- gDirectory (TDirectory) current directory
- gPad (TPad) current graphics pad
- gRandom (TRandom3 by default) random numbers
- … and some more

Global variables
gROOT

- gROOT keeps track of the complete root session and serves as the ROOT base directory
- Most objects created in root are accessible by lists managed by gROOT

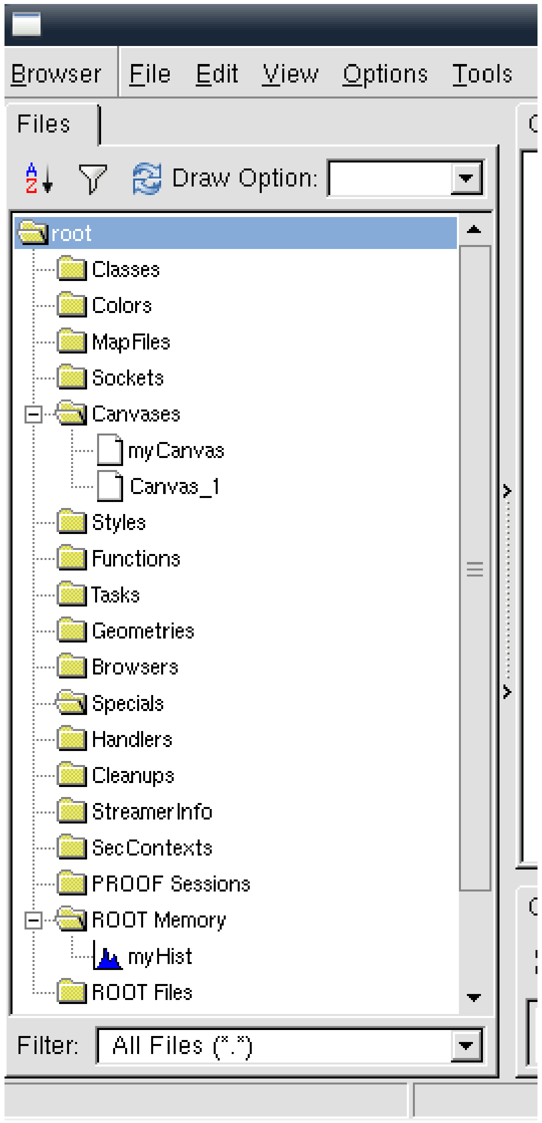
  http://root.cern.ch/root/html/TROOT.html

TSeqCollection* fFiles List of TFile
TSeqCollection* fMappedFiles List of TMappedFile
TSeqCollection* fSockets List of TSocket and TServerSocket
TSeqCollection* fCanvases List of TCanvas
TSeqCollection* fStyles List of TStyle
TSeqCollection* fFunctions List of TF1, TF2, TF3
TSeqCollection* fTasks List of TTask
TSeqCollection* fColors List of TColor
TSeqCollection* fGeometries List of geometries
TSeqCollection* fBrowsers List of TBrowser
TSeqCollection* fSpecials List of special objects
TSeqCollection* fCleanups List of recursiveRemove collections

Global variables

gROOT – examples

```
root [0] TCanvas *c=new TCanvas("myCanvas","My Canvas");
root [1] c
(class TCanvas*)0x1858550
root [2] gROOT->GetListOfCanvases()-
>FindObject("myCanvas")
(const class TObject*)0x1858550
root [3] TH1F *h=new TH1F("myHist","histo",100,0,100);
root [4] gROOT->FindObjectAny("myHist")
(const class TObject*)0x197bc60
root [5] gROOT->FindObjectAny("myCanvas")
(const class TObject*)0x1858550
```



```
root
  Classes
  Colors
  MapFiles
  Sockets
  Canvases
    myCanvas
    Canvas_1
  Styles
  Functions
  Tasks
  Geometries
  Browsers
  Specials
  Handlers
  Cleanups
  StreamerInfo
  SecContexts
  PROOF Sessions
  ROOT Memory
    myHist
  ROOT Files
```

Global variables

gStyle

- gStyle can be used to globally set a style for the graphics, such as all attributes from the **Att** classes
- Additionally e.g. what is displayed in the statistics box can be set, such as mean, rms, fit values, …
- Also the positions of the titles and statistics box, if a grid is shown in the pad, the font style and size are configurable
- For all possibilities check:

  http://root.cern.ch/root/html/TStyle.html

Global variables

gStyle – examples

Example of rootlogon.C:

```
void rootlogon()
{
  const Int_t NCont=255;
  TStyle *st = new TStyle("mystyle","mystyle");
  gROOT->GetStyle("Plain")->Copy((*st));
  st->SetTitleX(0.1);
  st->SetTitleW(0.8);
  st->SetTitleH(0.08);
  st->SetStatX(.9);
  st->SetStatY(.9);
  st->SetNumberContours(NCont);
  st->SetPalette(1,0);
  st->SetOptStat("erm");
  st->SetOptFit(0);
  st->SetGridColor(kGray+1);
  st->SetPadGridX(kTRUE);
  st->SetPadGridY(kTRUE);
  st->SetPadTickX(kTRUE);
  st->SetPadTickY(kTRUE);
  st->SetMarkerStyle(20);
  st->SetMarkerSize(.5);
  st->cd();
}
```

This sets gStyle to the style 'st'

Example of .rootrc:

```
Rint.Logon: $(HOME)/rootlogon.C
```

Important:

Make sure there is...

- **NO space at the end of the line**
- **an empty new line after that line**

Otherwise, it won't work at all or in some cases it only works if you start root from your home directory

Global variables

gSystem

- gSystem provides access to the underlying OS
- E.g. it is possible to list directory contents, check for file existence, execute shell commands
- Useful if lists of files should be created inside the code
- For details see:

  http://root.cern.ch/root/html/TSystem.html

Global variables

gSystem – examples

```
// check if a file exists, returns 'kFALSE'
// if the file exists and 'kTRUE' in case is does not
// the logic is inverse ...
Bool_t exists=gSystem->AccessPathName("/tmp/noFile");
if (exists==kTRUE) cout << "File '/tmp/noFile' does not exist!" << endl;

Bool_t exists=gSystem->AccessPathName("gSystem_examples.txt");
if (exists==kFALSE) cout << "File 'gSystem_examples.txt' exist!" << endl;

// my favourite way of using things -- but this only works for
// linux systems -- is to directly execute linux commands and
// read back the stdout output
// e.g. list all files in the '/tmp' directory
TString result=gSystem->GetFromPipe("ls /tmp");
TObjArray *arr=result.Tokenize("\n");
for (Int_t i=0; i<arr->GetEntriesFast(); ++i) cout << arr->At(i)->GetName() << endl;

// or line by line display the contents of this file:
TString result=gSystem->GetFromPipe("cat gSystem_examples.txt");
TObjArray *arr=result.Tokenize("\n");
for (Int_t i=0; i<arr->GetEntriesFast(); ++i) cout << arr->At(i)->GetName() << endl;
```

gSystem_examples.txt

More classes and name spaces

Linear algebra

- TVectorT<float/double> → TVectorF, TVectorD
- TMatrixT<float/double> → TMatrixF, TMatrixD
- Those classes can be used for linear algebra, matrix vector multiplication, vector – vector adding, multiplication, etc.
- The vector classes can also be used as simple replacement for c-arrays

http://root.cern.ch/root/html/TVectorT_float_.html
http://root.cern.ch/root/html/THilbertMatrixT_float_.html

More classes and name spaces

Histograms with arbitrary dimensions

- THnSparseT<...> – a multi-dimensional histogram where only bins for existing data will be created
- THnT<...> – a multi-dimensional histogram where all bins will be filled
- Both histograms are potential memory leaks, since it is easy to crate arbitrary dimensions with many bins. Think about the size the object might have before creating it

More classes and name spaces

More graphics functions

- To add graphical objects to the canvas many classes exist e.g.
  - TArrow, TLine, TPolyLine, TPave, TPaveText, TLatex, …
- For examples see:

$ROOTSYS/tutorials/graphics/
http://root.cern.ch/root/html534/tutorials/graphics/index.html

Coding Conventions

At this time you probably have already noticed some of them :

Classes	begin with T	TH1, TF1, TBrowser
Non-class types	end with _t	Int_t
Data members	begin with f	fTree
Member functions	begin with a capital	Loop()
Constants	begin with k	kRed
Non-local variables	begin with g	gEnv

exercises

Exercise 1: save own class into a ROOT file

- Take the example “classes” from lecture1
- Modify the class Point2D so that it can be saved in a ROOT file
- Rename the main in the file with another name, and modify it so that it saves an object Point2D in a ROOT file
- Compile the code from the ROOT prompt using “.L namefile.cpp++” and after this, call the function

- After the ROOT file is created, you can open it from the ROOT prompt and use the object saved. Remember to load the shared library before

  solution: exercises/day05/macros/classesAndIO.cpp

Exercise 1: save own class into a ROOT file step by step

- Take the example “classes” from lecture1
- Modify the class Point2D so that it can be saved in a ROOT file:
  - Add inheritance from TObject
  - Add ClassDef statement
  - Check that the default constructor satisfy the requirements of initializing all members. If not, modify it.
- Rename the main in the file with another name, and modify it so that it saves an object Point2D in a ROOT file
  - Change the name of the main (this is needed for the execution in the ROOT prompt)
  - create an object of type Point2D and initialize it with same values
  - open a ROOT file for writing
  - Save the object (Note: use the Write(..) function derived from TObject, passing as argument the name you want to assign to you object)
- Compile the code from the ROOT prompt using “.L namefile.cpp++” and after this, call the function
- After the file is created, you can open it from the ROOT prompt and use the object saved. Remember to load the shared library before
  - load the library using ; gSystem → Load (...proper name of the library…)
  - Open the file and access the object

Exercise 2: Use classes in a shared library in your macro

- Create a macro that generates events of type myEvent and particles of type myParticle, fill a tree with the events and save it to a file

- NB: to compile and execute the macro, from the ROOT prompt:

  - load the library libMyPackage.so

  - Use gSystem→AddIncludePath("…") to include the proper path to the code of myEvent and myParticle

  - compile your macro: .L yourmacro.C++

  - Execute it : yourmacro(..)

solution: exercises/day05/macros/simpleGenerate.C

Exercise 2: Use classes in a shared library in your macro step by step

Create a macro that generates events and particles, fill a tree and save it to a file

- Include the proper headers (all the ROOT classes used, myEvent and myParticle)
- Create a function that:
  - Open a ROOT file to write
  - Create a TTree to store myEvent objects
  - Makes a loop over N events (N could be an argument of the function or a fixed number) and inside the loop:
    - fill the event (Look the header of myEvent: Which members does the myEvent hold ? Which functions to set a value into the member should you use ?)
    - To fill the particles, make another loop over N2 particles inside the one on the events:
      - As before: Look the header of myParticle: Which members does the myParticle hold ? Which functions to set a value into the member should you use ?
      - Close the loop on the particles
    - Fill the TTree
    - Clear the event before continuing the loop on the events – see which function of myEvent you should use
- Save the TTree to the file and close it

Implement very small pieces, compile it and test it. If OK, go ahead. Do not implement all in one go!!

Extra development

- You may think to extend this exercise as you wish:
- Extend the 'myParticle' class. What kind of information does a particle need?

  After your modification, you should recreate the shared library using “make”
- Add new classes : you should add them into the Makefile, then recreate the shared library

Extra exercises on old material

TH2 Projections

- Create a 2 dimensional histogram

  Fill the histogram with correlated random normal numbers. To do this generate 2 random normal numbers (mean=0, sigma=1) u and w. Then use x = u and y=w+0.5*u for filling the histogram.
- Make projection histograms on one axis for different bin ranges of the other within a loop (Hint: look for TH1F::ProjectionX(..), TH1F::ProjectionY(..) function)
- Fill a TGraph with the mean values of the projected histograms vs the mean value of the bin range

Comparison of fits on histograms with different statistics

- Fill N histograms with numbers following a random Gaussian distribution, using always same mean and sigma. Start with a large number of entries for histogram (e.g. 10000)
- Fit the histograms and take the mean and the error from the fits
- Fill two histograms with these values: one containing the mean values from the fit, the other the errors from the fit

- Repeat the generation and the fits few times, changing the number of entries for histogram

- At the end you will have histograms containing the mean values and error for different statistics (that is, the number of entries used in the Gaussian generation). Compare them in the way you think is more appropriate (e.g. : draw the histograms together, get their mean and plot it vs the number of entries….)

  Are the results as expected ?

  NB: there are many ways to solve this exercise, more or less optimised …

  try to write a macro (better compiled) with different functions that perform the different tasks.