Building the Basic Structure For An Stntuple Analysis (Version 1)

(CDF Collaboration)

Jason Nett

(Dated: 13 September 2008)

The motivation for this note grew out of my desire for it to have already existed as I setup my own Stntuple analysis modules. Unlike the pen and paper study of physics, much of the structure of computing must simply be provided. The answer to a problem is often not an idea that can be logically deduced; rather, it's a result of an aribrary choice made by a person to 'make it work.' As such, there is a need for a comprehensive source of information that allows the novice to produce a working structure for an Stntuple analysis module. I provide here step-by-step instructions to start from scratch and end with an empty, though functioning, Stntuple module. We will see as we work through the minutia that doing so requires many seemingly arbitrary steps indeed.

Following this document step-by-step, the reader should be able to arrive at a skeleton Stntuple analysis including an user-built ntuple, a module that fills this ntuple with information from an Stntuple dataset, and an analysis module that is capable of processing the information stored in the ntuple. All that is required to begin is a home directory, access to the CDF software framework, and proper access to Stntuple data files.

Contents

DISCLAIMER

The approach I describe below should in no way be interpreted as being the *only* method to produce an functioning Stntuple module. Also, if any mistakes are found and need correcting, if any necessary information is missing, if someone wants to contribute further exposition explaining a topic, or if someone comes across (and solves) a nasty error that others might also impale themselves on, please email Jason Nett at jnett@wisc.edu. Nevertheless, I do not mearly attempt to describe the necessary steps in a disjointed manner; rather, the following should at least be sufficient to produce a function, compiling, running module inheriting from `TStnModule`, an ntuple module that the previous module fills with Stntuple data, and an analysis module capable of retrieving information from the ntuple and further processing it in some way. Every necessary command is given explicitly.

THE FIRST COMPILATION

Creating a New Release and its Directories

[**NOTE: DO NOT ATTEMPT TO COMPILE UNTIL THE "COMPILE" SECTION. YOU WILL RECEIVE ERRORS UNTIL ALL PREVIOUS SECTIONS ARE COMPLETED.**]

We begin as we do for so many tasks, breath life into a test release.

```
setup cdfsoft2 6.1.4
newrel -t 6.1.4 stn_example
cd stn_example
```

Next, we remove the environmental variable `USESHLIBS` [1].

```
unsetenv USESHLIBS
```

We also require a copy of the Stntuple package. This usually takes at least 10-20 minutes to compile.

```
addpkg Stntuple dev_243
gmake Stntuple.dev_243 MODE=reading
gmake Stntuple._ana USESHLIBS=1
```

Many new folders just appeared after compiling Stntuple. Most of these can be ignored for now. The `rootlogon.C` file is important for loading the shared libraries to `ROOT` and addressed below.

If we `cd include` we see that several symbolic links have appeared where before there were none. We will be adding one of our own by hand later.

I've found that if working in `tsch`, as opposed to `bash` shell, the `USESHLIBS=1` add-on is critically important during compilation of one of the branches of Stntuple or the module we will be creating shortly. Without it, the necessary shared libraries that are loaded into `ROOT` are not available in an updated form. When working in `bash` shell, it is not necessary.

Now we need to create the folders in which our own analysis will be conducted. There is a conventional method for how to construct these directories and what to put in them that is not necessarily critical for functionality, but I will follow the convention anyways. First, create a folder for our personal analysis and enter it

```
mkdir MyAnalysis
cd MyAnalysis
```

Second, create the following directories:

```
mkdir MyAnalysis
mkdir src
mkdir dict
mkdir drivers
mkdir data
```

`MyAnalysis` will contain the header file of the module `MyModule` that we will create shortly. `src` will contain the source code of `MyModule`. `dict` will contain a file called `MyModule_linkdef.h` that is critical for compilation. `drivers` will contain the `ROOT` script that controls the execution of our module. `data` is the directory where we will put a sample Stntuple data set that our analysis module will access. All these are explored in further detail below.

The Source Code and Header Files

Let's enter the directory \ `stn_example` \ `MyAnalysis` \ `src` \ . Create a file to contain our source code:

```
xemacs MyModule.cc &
```

Similarly, in the directory \ `stn_example` \ `MyAnalysis` \ `MyAnalysis` \ create the header file for our new module:

```
xemacs MyModule.hh &
```

Let's begin filling in some basic structure for the header file (beware the dangers of blindy copy and pasting; differences can creep in):

```
#if !defined (__CINT__) || defined (__MAKECINT__)

#include "TH1.h"
#include "TH2.h"
#include "TProfile.h"
#include <TPostScript.h>
#include <Stntuple/loop/TStnModule.hh>
#include <Stntuple/obj/TCalDataBlock.hh>
#include <Stntuple/obj/TStnTriggerBlock.hh>
#include <Stntuple/obj/TStnPhotonBlock.hh>
#include <Stntuple/obj/TStnJetBlock.hh>
#include "Stntuple/obj/TStnJetProbBlock.hh"
#include <Stntuple/obj/TStnElectronBlock.hh>
#include <Stntuple/obj/TStnMuonBlock.hh>
#include <Stntuple/obj/TStnMetBlock.hh>
#include <Stntuple/obj/TStnDBManager.hh>
#include <Stntuple/obj/TStnRunSummary.hh>
#include <Stntuple/obj/TStnTrackBlock.hh>
#include <Stntuple/obj/TStnVertexBlock.hh>
#include <Stntuple/obj/TCesDataBlock.hh>
#include <Stntuple/obj/TStnClusterBlock.hh>
#include <Stntuple/obj/TDcasDataBlock.hh>
#include <Stntuple/obj/TPhoenixElectronBlock.hh>
#include <Stntuple/obj/TGenpBlock.hh>
#include <Stntuple/obj/TSvtDataBlock.hh>
#include <Stntuple/alg/TStntuple.hh>
#include <Stntuple/obj/TStnSecVtxTagBlock.hh>
#include <Stntuple/obj/TStnTriggerTable.hh>
#include <Stntuple/obj/TStnEvent.hh>

#endif

#include <map>
#include <set>
#include <vector>
#include <iostream>

using std::map;
using std::set;
using std::vector;
using std::cout;
using std::endl;

class MyModule: public TStnModule
{
  public:
   string mode;
   ~MyModule(){}
   MyModule();

   // Re-implement the TStnModule methods
   int     BeginJob();
   int     BeginRun();
   int     Event(int ientry);
   int     EndJob();
  protected:
};
```

In the source code write:

```
#include "TF1.h"
#include "TCanvas.h"
#include "TLine.h"
#include "TText.h"
#include "TMath.h"
#include "TStyle.h"
#include "TChain.h"
#include "TRandom.h"
#include <set>
#include <iostream>

using namespace std;
// if the following line doesn't work, see the symbolic link section
#include "MyAnalysis/MyModule.hh"
#include "Stntuple/obj/TStnTriggerBlock.hh"
#include "Stntuple/obj/TStnHeaderBlock.hh"
#include "Stntuple/obj/TStnTrackBlock.hh"
#include "Stntuple/obj/TStnPhotonBlock.hh"
#include "Stntuple/obj/TStnElectronBlock.hh"
#include "Stntuple/obj/TStnJetBlock.hh"
```

```
#include "Stntuple/obj/TStnMuonBlock.hh"
#include "Stntuple/obj/TStnMetBlock.hh"
#include "Stntuple/obj/TStnRunSummary.hh"
#include "Stntuple/loop/TStnAna.hh"
#include "Stntuple/loop/TStnInputModule.hh"
#include <Stntuple/obj/TStnNode.hh>
//Constructor
MyModule::MyModule():TStnModule("MyModule","MyModule"){}

int MyModule::Event(int ientry)
{
  std::cout << "MyModule::Event" << std::endl;
  return 0;
}

int MyModule::BeginJob()
{
  return 0;
}

int MyModule::BeginRun()
{
  return 0;
}

int MyModule::EndJob()
{
  return 0;
}
```

The first thing to note is that this module inherits from `TStnModule`. So if we go to the top directory of this release and open `TStnModule.cc`

```
xemacs Stntuple/loop/TStnModule.cc &
```

we find empty functions by the same name as above. These in turn are called by `TStnAna`, which controls the event loop. Notice that each of these functions has a `return 0;` which must be the last line of the function. If execution does not reach this line, then `TStnAna` knows that something has gone wrong and will print an error. See the section for the driver script for more details.

Any Stntuple analysis module will require the ability to access data from the Stntuple data blocks and will utilize the usual beam position corrections for each run of data input. The following two subsections describe this setup.

MyModule::BeginJob()

In the `public:` section of the header file write:

```
// StnTuple Data blocks
TStnElectronBlock*      fElectronBlock;
TStnMuonBlock*          fMuonBlock;
TStnMetBlock*           fMetBlock;
TStnJetBlock*           fJetBlock;
TStnJetProbBlock*       fJetProbBlock;
TCalDataBlock*          fCalData;
TStnTrackBlock*         fTrackBlock;
TCesDataBlock*          fCesDataBlock;
TStnClusterBlock*       fClusterBlock;
TStnPhotonBlock*        fPhotonBlock;
TStnVertexBlock*        fVertexBlock;
TDcasDataBlock*         fDcasDataBlock;
TPhoenixElectronBlock*  fPhoenixElectronBlock;
TStnTrackBlock*         fPhoenixSeedTrackBlock;
TStnTrackBlock*         fPhoenixSiTrackBlock;
TGenpBlock*             fGenpBlock;
TStnTriggerBlock*       fTriggerBlock;
TSvtDataBlock*          fSvtDataBlock;
TStnTriggerTable*       ftrigtable;
TStnSecVtxTagBlock*     fSecVtxTag;
TStnRunSummary*         runSummary;
```

Within `MyModule::BeginJob` of the source code write:

```
TString name = GetName();
if (name.Contains("data") || name.Contains("DATA") || name.Contains("Data")) mode ="data";
cout << "                                          name = " << name << endl;
cout << "                                          mode = " << mode << endl;

fElectronBlock         = (TStnElectronBlock*)     RegisterDataBlock("ElectronBlock","TStnElectronBlock");
fMetBlock              = (TStnMetBlock*)          RegisterDataBlock("MetBlock","TStnMetBlock");
fMuonBlock             = (TStnMuonBlock*)         RegisterDataBlock("MuonBlock","TStnMuonBlock");
fCalData               = (TCalDataBlock*)         RegisterDataBlock("CalDataBlock","TCalDataBlock");
```

```
fJetBlock              = (TStnJetBlock*)          RegisterDataBlock("JetBlock","TStnJetBlock");
fJetProbBlock          = (TStnJetProbBlock*)      RegisterDataBlock("JetProbBlock","TStnJetProbBlock");
fDcasDataBlock         = (TDcasDataBlock*)        RegisterDataBlock("DcasDataBlock","TDcasDataBlock");
fClusterBlock          = (TStnClusterBlock*)      RegisterDataBlock("ClusterBlock","TStnClusterBlock");
fCesDataBlock          = (TCesDataBlock*)         RegisterDataBlock("CesDataBlock","TCesDataBlock");
fPhotonBlock           = (TStnPhotonBlock*)       RegisterDataBlock("PhotonBlock","TStnPhotonBlock");
fTrackBlock            = (TStnTrackBlock*)        RegisterDataBlock("TrackBlock","TStnTrackBlock");
fVertexBlock           = (TStnVertexBlock*)       RegisterDataBlock("ZVertexBlock","TStnVertexBlock");

if (mode=="data")
  fTriggerBlock          = (TStnTriggerBlock*)      RegisterDataBlock("TriggerBlock", "TStnTriggerBlock");
else
  fTriggerBlock          = (TStnTriggerBlock*)    RegisterDataBlock("TrigSimBlock", "TStnTriggerBlock");

fPhoenixSiTrackBlock   = (TStnTrackBlock*)        RegisterDataBlock("PROD@PhoenixSI_Tracking","TStnTrackBlock");
fPhoenixElectronBlock  = (TPhoenixElectronBlock*) RegisterDataBlock("Phoenix_Electrons","TPhoenixElectronBlock");
fPhoenixSeedTrackBlock = (TStnTrackBlock*)        RegisterDataBlock("PROD@Phoenix_Tracking","TStnTrackBlock");
fGenpBlock             = (TGenpBlock*)            RegisterDataBlock("GenpBlock","TGenpBlock");
fSvtDataBlock          = (TSvtDataBlock*)         RegisterDataBlock("SvtDataBlock","TSvtDataBlock");
fSecVtxTag             = (TStnSecVtxTagBlock*)    RegisterDataBlock("SecVtxTagBlock","TStnSecVtxTagBlock");

// check for datablocks that are null
TObjArray* nodes = GetAna()->GetEvent()->GetListOfNodes();

for (int i=0; i< nodes->GetEntries(); i++)
  {
    TStnNode* n = (TStnNode*) nodes->At(i);
    cout << (n->GetBranch()->GetName()) << " at " <<  n->GetDataBlock() << endl;
    if (!n->GetDataBlock())
      {
        cout << " Error : Did not find Branch " << n->GetBranch()->GetName() << endl;
        exit(0);
      }
  }
return 0;
```

Of course, `return 0;` should only appear once in each function.

MyModule::BeginRun()

In practice, the beamline is constantly shifting within some small area of a plane perpendicular to it. This exact location of the beamline shifts from one run to the next as well as within the execution of a run. As such, the timing information of particles emitted from the collisions in the beam require precise knowledge of where the beam is. Taking this into account is the purpose of the following code. In the public section of the header file, write:

```
std::set<std::string> inputfiles;
TStnBeamPos* beampos_cot;
TStnBeamPos* beampos_svx;
```

Within `MyModule::BeginRun` of the source code, write:

```
// Get the trigger table for this run.
TStnDBManager* dbm = TStnDBManager::Instance();
ftrigtable = (TStnTriggerTable*) dbm->GetTable("TriggerTable");
beampos_cot = (TStnBeamPos*) dbm->GetTable("CotBeamPos");
beampos_svx = (TStnBeamPos*) dbm->GetTable("SvxBeamPos");
// maintain a list of all input files
TChain* c = GetAna()->GetInputModule()->GetChain(); //get get get get get get
inputfiles.insert(string(c->GetFile()->GetName())); //(we love OO programming)
TStnRunSummary* rs = (TStnRunSummary*) dbm->GetTable("RunSummary");
return 0;
```

Symbolic Linking and the Header

Recall the line

```
#include "MyAnalysis/MyModule.hh"
```

in the source code. It would not work without the following.

Go back to the top directory of the release and enter the `include` folder. Once there, type

```
ln -s ~myhomedirectory/stn_example/MyAnalysis/MyAnalysis MyAnalysis
```

to create what's called a "symbolic link" to the header file's directory. This will allow the source code to know where to look for its header file.

If this is done properly, you should be able to type `ls -all` and see a line similar to:

```
lrwxrwxrwx  1 jnett80  cdf  48 Jul 24 00:30 MyAnalysis -> /home/jnett80/stn_example/MyAnalysis/MyAnalysis/
```

MyModule_linkdef.h

Go to the directory \ `stn_example` \ `MyAnalysis` \ `dict` \. Create a file:

```
xemacs MyModule_linkdef.h
```

It is very important to follow the naming convention. Within this file type:

```
#ifdef __CINT__
#pragma link off all   globals;
#pragma link off all   classes;
#pragma link off all   functions;
#pragma link C++ class MyModule;
#endif
```

For more complex analyses that contain many modules, it is customary to have a separate `*_linkdef.h` file for each class. We will see examples in the "The First Tasks" section when the construction of ntuples is addressed.

The GNUmakefile files

The control of compilation is guided by the `GNUmakefile`'s, and several are required. There should already exist a `GNUmakefile` in the top directory of the release containing simply:

```
include SoftRelTools/GNUmakefile.main
```

Next, change directories to our analysis folder `cd MyAnalysis`. There should not already be a `GNUmakefile` here so we make one: `xemacs GNUmakefile &`. Copy into this:

```
# Makefile for Examples package
#
# uses SoftRelTools/standard.mk
#
#################################################################

# subdirectories
SUBDIRS = src dict

ifdef USESHLIBS
  LINK_SHARED = 1
endif

export LINK_SHARED

#################################################################
include SoftRelTools/standard.mk
```

At the moment, we only have code that requires compilation in the folders `src` and `dict`. Later in your analysis you will likely have other folders containing modules that require compilation. At that point, the names of the folders that contain them will have to be added to the `SUBDIRS` line above.

Change directories to the `src` folder and create another file with the same name: `xemacs GNUmakefile &`. Put in this file:

```
# Makefile for Examples package
#
# uses SoftRelTools/standard.mk
#
#############################################################
# file lists (standard names, local contents)

# include file products
INC =

# library product
```

```
ifdef LINK_SHARED
  SHAREDLIB = libMyAnalysis.so
else
  LIB = libMyAnalysis.a
endif

LIBCCFILES = $(filter-out $(skip_files), $(wildcard *.cc))

# subdirectories
SUBDIRS =

BINS =

include PackageList/link_all.mk

############################################################
include SoftRelTools/standard.mk
include SoftRelTools/arch_spec_root_minedm.mk
include SoftRelTools/refresh.mk
```

It is critically important to follow the naming convention for the library names. The name must start with `lib`, then the name of the analysis folder (not the name of the module), then end with `.so` or `.a`.

Lastly, go to the `dict` folder and create yet another `GNUmakefile`. In this one, put

```
LINK_SHARED_MODULES = yes
ifdef LINK_SHARED
  SHAREDLIB       = libMyAnalysis.so
else
  LIB             = libMyAnalysis.a
staticlib_o_dir = $(sharedlib_o_dir)
endif
lib_o_dir       = $(sharedlib_o_dir)

skip_files =
LIBCXXFILES = $(filter-out $(skip_files), $(wildcard $(lib_o_dir)*_dict.cxx))
#############################################################################
CINT_SUBDIRS =  $(SRT_TOP)/MyAnalysis:$(SRT_TOP)/MyAnalysis/MyAnalysis:$(SRT_TOP)/include/MyAnalysis/MyAnalysis

vpath %.hh .:$(SRT_TOP)/MyAnalysis/MyAnalysis
vpath %.h  .:$(SRT_TOP)/MyAnalysis/MyAnalysis

#############################################################################
#override SRT_QUAL := debug

include SoftRelTools/standard.mk
include SoftRelTools/refresh.mk

include SoftRelTools/arch_spec_root.mk
include RootUtils/arch_spec_rootcint.mk

lib: codegen
```

The empty `.refresh` file

This part is also critical for successful compilation, but also likely the most easily overlooked. In both the `src` and `dict` directories you need to create an empty file called `.refresh`

```
xemacs .refresh &
```

Once opened, simply hit the space bar once, delete the space, then hit save. This way, the file will exist, but be completely empty. The "." at the beginning of the file name leaves this file invisible to the usual `ls` command, but will show when `ls -all` is executed.

Compile

We have now complete all necessary steps to compile (but not yet run) our empty Stntuple module. Go to the top directory of the release and type

```
gmake MyAnalysis.nobin USESHLIBS=1
```

Doing so, I get the following output indicating a successful compilation:

```
[jnett80@nuwm08 ~/stn_example]$ gmake MyAnalysis.nobin USESHLIBS=1
<**nobin**> MyAnalysis
Refreshing libMyAnalysis.so
-- codegen --  CINT HH header for MyModule
```

```
Error in <MyModule>: MyModule inherits from TObject but does not have its own ClassDef
<**compiling**> MyModule.cc
<**building library**>
<**compiling**> MyModule_dict.cxx
<**building library**>
```

The error that appears is not as important as it sounds and not at the moment. We will deal with that in the "`ClassDef` and `ClassImp`: Making Your Class a ROOT Class" section below.

THE FIRST RUN

`rootlogon.C` and Loading `MyAnalysis`' shared library

If compilation was successfully accomplished, check to see that the shared library of our new analysis was generated. From the top directory of the release:

```
cd shlib/Linux2_SL-GCC_3_4/
```

Listing the files, I see:

```
libMyAnalysis.so    libStntuple_ana.so   libStntuple_geom.so  libStntuple_obj.so
libStntuple_alg.so  libStntuple_base.so  libStntuple_loop.so  libStntuple_val.so
```

So the shared library exists where it should.

Let's go back to the top directory of the release and open the `rootlogon.C` file. Upon entering `ROOT`, this file will execute first as though you were typing each line at the `ROOT` command prompt. Here is what I see:

```
//------------------------------------------------------------------------------
//  rootlogon.C: a sample ROOT logon macro allowing use of ROOT script
//               compiler in CDF RunII environment. The name of this macro file
//               is defined by the .rootrc file
//
//  USESHLIBS variable has to be set to build Stntuple libraries locally:
//
//  setenv USESHLIBS 1
//
//  Feb 12 2001 P.Murat
//------------------------------------------------------------------------------
{
#include <iomanip.h>
#include <time.h>
                                // the line below tells ROOT script compiler
                                // where to look for the include files

  gSystem->SetIncludePath(" -I./include -I$CDFSOFT2_DIR/include");

                                // load in ROOT physics vectors and event
                                // generator libraries

  gSystem->Load("$ROOTSYS/lib/libPhysics.so");
  gSystem->Load("$ROOTSYS/lib/libEG.so");
  gSystem->Load("$ROOTSYS/lib/libDCache.so");

  // load a script with the macros
  char command[200];

  sprintf(command,"%s/Stntuple/scripts/global_init.C",
    gSystem->Getenv("CDFSOFT2_DIR"));

  gInterpreter->LoadMacro(command);

                                // STNTUPLE shared libraries are assumed to be
                                // built in the private test release area with
                                // USESHLIBS environment variable set
                                // we always need libStntuple_loop, but the
                                // other 2 libs should be loaded in only if
                                // we're running bare root

  const char* exec_name = gApplication->Argv(0);

  if ((strstr(exec_name,"root.exe") != 0) || (strstr(exec_name,"stnfit.exe") != 0)) {
    gSystem->Load("./shlib/$BFARCH/libStntuple_base.so");
    gSystem->Load("./shlib/$BFARCH/libStntuple_obj.so");
    gSystem->Load("./shlib/$BFARCH/libStntuple_loop.so");
  }

  if (strstr(exec_name,"root.exe") != 0) {
    gSystem->Load("./shlib/$BFARCH/libStntuple_geom.so");
```

```
    gSystem->Load("./shlib/$BFARCH/libStntuple_alg.so");
    gSystem->Load("./shlib/$BFARCH/libStntuple_ana.so");
    gSystem->Load("./shlib/$BFARCH/libStntuple_val.so");
  }

                                // print overflows/underflows in the stat box
  gStyle->SetOptStat(11111111);
                                // print fit results in the stat box
  gStyle->SetOptFit(1110);
                                // this line reports the process ID which simplifies
                                // debugging

  TAuthenticate::SetGlobalUser(gSystem->Getenv("USER"));
  gInterpreter->ProcessLine(".! ps | grep root");
}
```

In the section:

```
if (strstr(exec_name,"root.exe") != 0) {
  gSystem->Load("./shlib/$BFARCH/libStntuple_geom.so");
  gSystem->Load("./shlib/$BFARCH/libStntuple_alg.so");
  gSystem->Load("./shlib/$BFARCH/libStntuple_ana.so");
  gSystem->Load("./shlib/$BFARCH/libStntuple_val.so");
}
```

add the line:

```
gSystem->Load(''./shlib/\$BFARCH/libMyAnalysis.so'');
```

This will load the shared library of our new analysis to ROOT along with the others.

Retriving a set of test data from SAM/Accessing SAM data directly

If we are to perform a successful run of our new module, we need some Stntuple data to run through it. There are several ways to access such data and if you already have an Stntuple file to use, most of this section can probably be ignored. I will provide one possible way to retrieve a data set. This will require access to the SAM directories, which requires SAM to be setup where you are working. I am working (remotely) from one of the Wisconsin group's computers at CDF, which does have access to SAM. Until recently, our local system did not have SAM, so this does not seem to be universal.

SAM files are organized first in "books;" each "book" contains "datasets" of some particular type; each "dataset" contains "filesets;" and each "fileset" contains data "files." Suppose SAM is available and your group's webpage has an attractive file called cc036b44.0001hgs1. You can put the file anywhere you like; I'm putting it in the directory \ stn_example \ MyAnalysis \ data \ by going to that directory and then executing the following lines:

```
setup sam
setup diskcache_i -q GCC_3_4_3
setup dcap
sam get dataset --fileList="cc036b44.0001hgs1" --group=test --downloadPlugin=dccp
```

After waiting a while for this set to download into the directory from which I executed the previous commands from I see the following output:

```
[jnett80@nuwm08 data]$ sam get dataset --fileList="cc036b44.0001hgs1" --group=test --downloadPlugin=dccp
HeavyConsumer: 07/24/08 12:47:33: INFO : Station is set to cdf-caf
HeavyConsumer: 07/24/08 12:47:33: INFO : Created definitation name def_jnett80_20080724124733
HeavyConsumer: 07/24/08 12:47:34: INFO : Created snapshot 689040
HeavyConsumer: 07/24/08 12:47:34: INFO : Defaulting timeout to 3600 seconds
ProjectInfo({
     'baseProjectInfo' : BaseProjectInfo({
        'personInfo' : PersonInfo({
     'emailAddress' : 'jnett@wisc.edu',
        'firstName' : 'None',
         'lastName' : 'None',
         'personId' : 1360L,
     'personStatus' : 'active',
              'uid' : '0',
         'userName' : 'jnett80',
     }),
         'projectId' : 1075035L,
       'projectMode' : 'unknown',
       'projectName' : 'jnett80_20080724124733',
     'projectStatus' : 'reserved',
        'snapshotId' : 689040L,
       'stationName' : 'cdf-caf',
     'workGroupName' : 'test',
     }),
            'endTime' : SamTime('NULL'),
           'nodeName' : 'fcdfsam1.fnal.gov',
```

```
            'osPID' : 5990L,
          'startTime' : SamTime(1216921655.0),
    })
Requesting first file...
dccp dcap://cdfdca3.fnal.gov:25155/pnfs/fnal.gov/usr/cdfen/filesets/NS/NS01/NS0106/NS0106.6/cc036b44.0001hgs1 .
1300878356 bytes in 294 seconds (4321.05 KB/sec)
Requesting next file...
No more files
```

The Driver Script

The last major step is to make a ROOT driver script. It is this script that will be called at the command line, access whatever data we want to run the module on, and run those modules.

In the directory \ stn_example \ MyAnalysis \ drivers \ make a file myDriver.C and put in it:

```
#include <iostream>
TStnAna*     x;
TChain*      inChain;
TStnCatalog* catalog;
TStnDataset* dataset;

void myDriver(int iset = 0, int NEvents=1000000)
{
  gStyle = new TStyle("Plain","Plain");
  inChain = new TChain("STNTUPLE");
  TString name="none";
  if (iset==10)
    {
      /*
      ACCESS A LOCALLY SAVED SET OF DATA
      */

      name ="WHlvbb";
      inChain->Add("~jnett80/stn_example/MyAnalysis/data/cc036b44.0001hgs1");
      x = new TStnAna(inChain);
    }

  if (iset == 20)
    {
      /*
      ACCESS A DATASET ON SAM DIRECTLY
      */

      // Name of the "dataset" on SAM we want to access
      char* datasetName = "chgs1c";

      // This value indicates we want to run over the 1st "fileset" in the "dataset"
      int segment = 1;

      catalog = new TStnCatalog();
      dataset = new TStnDataset();
      char istr[10];
      sprintf(istr, "%06i", segment);

      // "cdfpstn" is the name of the "book" the "dataset" is in
      catalog->InitDataset(dataset,"cdfpstn",datasetName,istr,"",0,999999);
      x = new TStnAna(dataset);
    }
  MyModule* m = new MyModule();
  m->SetName( name.Data() );
  x->AddModule(m);
  x->SetNEventsToReport(100000);
  x->Run(NEvents);
}
```

There are a few things to note. First, the name of the script and the function it contains are both myDriver. Next, it takes two arguments. The first, iset, is an option that allows easy access to different kinds of data if we have more than one type that we wish to separately run through our module. Lastly, I've noticed that the tilde (∼) tends to disappear when copy and pasting from this document. Be sure to include the tilde in front of the name of your home directory or the driver script will not find the data file.

In the case iset=10, we are running through our module the file that we saved to our local area in the previous section. In the case iset=20, we are directly accessing data on SAM to run through our module. For this, notice that we need only three pieces of information: the "book" name ("cdfpstn"), the "dataset" name ("chgs1c"), and the number of the "fileset" we wish to start running on. The "book" contains the "dataset", which contains "filesets", which contains "files", like the file we saved in the previous section. [2]

The second argument is the number of events to run through the module. In the driver script, it is important to understand that TStnAna is the Stntuple module that controls the event loop and runs the data through our module.

To run the script, go the the top level of our release and type:

```
root -l -q ''MyAnalysis/drivers/myDriver.C(10,10)'' >& log
```

which will run 10 events from the $WH \to l\nu b\bar{b}$ file acquired in the previous section from SAM. To run on SAM data directly, change the first argument to "20".

Run

Finally, we are ready for a first run of the data. When I execute the driver script as indicated in the previous section, I get:

```
11804 pts/2    00:00:00 root
11808 pts/2    00:00:01 root.exe

Processing MyAnalysis/drivers/myDriver.C(10,10)...
TStnRun2InputModule::BeginJob Warning - no metadata,
       opening all chained files to count entries...
TStnRun2InputModule::BeginJob: chained    1 files,     11540 events
                                           name = WHlvbb
                                            mode =
HeaderBlock at 0xba44110
ElectronBlock at 0xc4196d8
MetBlock at 0xc41d310
MuonBlock at 0xc41b138
CalDataBlock at 0xba44260
JetBlock at 0xc411f80
JetProbBlock at 0xc421148
DcasDataBlock at 0xc40fca0
ClusterBlock at 0xc41aac8
CesDataBlock at 0xba44bf0
PhotonBlock at 0xc41a7d0
TrackBlock at 0xc4110f0
ZVertexBlock at 0xc410f10
TrigSimBlock at 0xc40ea80
PROD@PhoenixSI_Tracking at 0xc411b00
Phoenix_Electrons at 0xc419db0
PROD@Phoenix_Tracking at 0xc4115c0
GenpBlock at 0xc418048
SecVtxTagBlock at 0xc41d468
MyModule::Event
MyModule::Event
MyModule::Event
MyModule::Event
MyModule::Event
MyModule::Event
MyModule::Event
MyModule::Event
MyModule::Event
MyModule::Event
----- end job: ---- StnAna
  L(TeV) :     -1.000
  L(live):     -1.000
  L(offl):     -1.000
  >>> TStnAna::EndJob: processed         10 events, passed         10 events
```

THE FIRST TASKS

`ClassDef` and `ClassImp`: Making Your Class a ROOT Class

Let's go one step further and make our class known to ROOT. There are a couple simple steps to take with one subtlety. First, it is necessary to have a constructor with no arguments–otherwise a cryptic error results. As we have already done this, let's move on to the header file. Add the following `ClassDef` line to the end of the class definition:

```
   ClassDef(MyModule,0)
};
```

In the source code, add the following `ClassImp` line after the `#include` statements:

```
#include <Stntuple/obj/TStnNode.hh>

//Inform ROOT of this class
ClassImp(MyModule)

//Constructor
MyModule::MyModule():TStnModule("MyModule","MyModule")
```

Now go back to the top directory of this release to recompile and run as before. See whatever ROOT manual is current for more information on ClassImp and ClassDef.

Declaring Static Data Members

This section may seem to be soley a C++ issue rather than an Stntuple issue, but it can be subtle to organize properly and is also highly relevant to subsequent topics. Many of the errors resulting from improper implementation are not caught by the compiler, but rather at run time. The resolution of such errors can, therefore, be difficult to rectify.

Static data memebers belonging to a class are sometimes also denoted "class variables." These are declared within the class declaration itself and have a unique value for all instances of the class. There are a myriad of reasons one might want to include such a member, but it's important to note that there are two steps to creating one. [3] First, we declare the variable under the public: section of the header file MyModule.hh.

```
// Class variable declaration
static const Int_t x;
```

Second, since header files do not like value-assignment (headers are usually meant only for declarations while implementation is reserved for the source code) we assign the value of a class variable within the source code MyModule.cc, but in a manner that the value-assignment is executed once and all of the source code is within it's scope. So we actually do not put the assignment inside any function; rather, we customarily assign the value just above the constructor as follows:

```
#include "Stntuple/loop/TStnInputModule.hh"
#include <Stntuple/obj/TStnNode.hh>

// Class variable value assignment
const int MyModule::x = 1;

//Constructor
MyModule::MyModule():TStnModule("MyModule","MyModule"){}
```

Access The Data Blocks

In the "Source Code and Header Files" section earlier we included and declared many Stntuple data blocks in the header and called RegisterDataBlock during MyModule::BeginJob for each of them. This "registers" the data blocks for our module, but does not fill them. It may have seemed a bit extraneous at the time, but now we'll make use of them.

Recall that MyModule::BeginJob executes once before any data is run through our module, but the information must obviously change for every event–no two events are the same. Hence, we must fill these blocks with event information for every event in the loop. Let's create a new function within MyModule that fill the blocks from MyModule::Event.

Let's begin with an example of fill a single data block, the TStnMuonBlock, and later replace it with a loop over all the data blocks. First, write into the public: section of MyModule.hh

```
// Fill the Stntuple data blocks
   void FillDataBlocks(int ientry);
```

Then write the body of this new function in the source code:

```
// Fill Stntuple Data Blocks
void MyModule::FillDataBlocks(int ientry)
{
   fMuonBlock ->GetEntry(ientry);
}
```

And, finally, we must call this function from MyModule::Event:

```
int MyModule::Event(int ientry)
{
   std::cout << "MyModule::Event" << std::endl;

   // Fill Stntuple Data Blocks
   FillDataBlocks(ientry);

   // Access the number of muons recorded for this event.
   int nmuons=fMuonBlock->NMuons();//returns the number of muons for this event
   std::cout << "  nmuons = " << nmuons << std::endl;
   return 0;
}
```

After recompiling and running you should see something like:

```
 7506 pts/3    00:00:00 root
   7507 pts/3    00:00:01 root.exe

Processing MyAnalysis/drivers/myDriver.C(10,10)...
TStnRun2InputModule::BeginJob Warning - no metadata,
      opening all chained files to count entries...
TStnRun2InputModule::BeginJob: chained    1 files,     11540 events
MyModule::BeginJob
 name = WHlvbb
 mode =
HeaderBlock at 0xba44330
ElectronBlock at 0xc4198f8
MetBlock at 0xc41d530
MuonBlock at 0xc41b358
CalDataBlock at 0xba44480
JetBlock at 0xc4121a0
JetProbBlock at 0xc421368
DcasDataBlock at 0xc40fec0
ClusterBlock at 0xc41ace8
CesDataBlock at 0xba44e10
PhotonBlock at 0xc41a9f0
TrackBlock at 0xc411310
ZVertexBlock at 0xc411130
TrigSimBlock at 0xc40eca0
PROD@PhoenixSI_Tracking at 0xc411d20
Phoenix_Electrons at 0xc419fd0
PROD@Phoenix_Tracking at 0xc4117e0
GenpBlock at 0xc418268
SecVtxTagBlock at 0xc41d688
MyModule::BeginRun
MyModule::Event
   nmuons = 1
MyModule::Event
  nmuons = 1
MyModule::Event
   nmuons = 0
MyModule::Event
   nmuons = 0
MyModule::Event
   nmuons = 2
MyModule::Event
   nmuons = 1
MyModule::Event
   nmuons = 3
MyModule::Event
   nmuons = 2
MyModule::Event
   nmuons = 6
MyModule::Event
   nmuons = 3
----- end job: ---- StnAna
  L(TeV) :     -1.000
  L(live):     -1.000
  L(offl):     -1.000
MyModule::EndJob
 >>> TStnAna::EndJob: processed         10 events, passed         10 events
```

To fill all the data blocks in a compact form, replace the line in `MyModule::FillDataBlocks` with

```
TObjArray* nodes = GetAna()->GetEvent()->GetListOfNodes();
for (int i=0; i< nodes->GetEntries(); i++)
  {
    TStnNode* n = (TStnNode*) nodes->At(i);
    n->GetDataBlock()->GetEntry(ientry);
  }
```

All we've actually accessed so far is the number of muons appearing in an event. Let's go a step further and see the kind of information we can learn about these muons. To see what stubs each muon has, we can loop over all the muons for each event. We can create a pointer to the instance of `TStnMuon` for each muon and access a function that returns a boolean for whether this muon has a particular stub. Add the following loop to `MyModule::Event`:

```
int MyModule::Event(int ientry)
{
  std::cout << "MyModule::Event" << std::endl;

  // Fill Stntuple Data Blocks
  FillDataBlocks(ientry);

  // Access the number of muons recorded for this event.
  int nmuons=fMuonBlock->NMuons();//returns the number of muons for this event
  std::cout << "  nmuons = " << nmuons << std::endl;
```

```
  for(int imu=0; imu<nmuons; imu++) //loop over the reconstructed muons
    {
      //Access reconstructed muons information
      TStnMuon* muo =  fMuonBlock->Muon(imu);//fMuonBlock is a TStnMuonBlock.
      TStnTrack* trk = fTrackBlock->Track(muo->TrackNumber());
      const bool cmupstub  = (muo->HasCmuStub() && muo->HasCmpStub());
      const bool cmxstub   = (muo->HasCmxStub());
      const bool cmpstub   = (muo->HasCmpStub() && !trk->IsCMUFid());
      const bool cmustub   = (muo->HasCmuStub() && !trk->IsCMPFid());
      const bool bmustub   = (muo->HasBmuStub() );
      std::cout << "    cmupstub = " << cmupstub << std::endl;
      std::cout << "    cmxstub  = " << cmxstub << std::endl;
      std::cout << "    cmpstub  = " << cmpstub << std::endl;
      std::cout << "    cmustub  = " << cmustub << std::endl;
      std::cout << "    bmustub  = " << bmustub << std::endl;
      std::cout << "    ---------------" << std::endl;
    }
  return 0;
}
```

Recompile, run, and check the log to see a listing of what stubs each muon has.

Make and Fill a Histogram

Histograms are centrally important for any kind of study. Now that we know how to access the information of an event stored in the Stntuple, let's generate a couple histograms of the output. It is important to create a particular histogram object once, then continually fill it for each event. This means we must instantiate our histogram class of choice outside of the event loop. One way to do this is declaring the histogram objects as static members.

Let's begin by choosing to work with the `TH1F` histogram class. Since this is a `ROOT` histogram such an object is useless unless we actually have a `ROOT` file to save it in. Hence, we also need a `TFile` object. In the class declarations of the header file `MyModule.hh`, write

```
class TFile;
class TH1F;
```

(this goes just above `class MyModule:  public TStnModule`).

Since we have just accessed the muon block of this Stntuple, let's make histograms of the η and the P_T distributions. In the `public:` section of the class declaration for `MyModule`, write

```
// Histograms
static TH1F* testEtaHistogram;
static TH1F* testPtHistogram;
static const Double_t etabins[41];
static const Double_t ptbins[151];
```

Declare the `TFile` object under the `protected:` section:

```
TFile*   _file;
```

That's it for the header file, now move on to the source code. Recall that we assign value to static members outside of any function, customarily just above the constructor:

```
// Class constants
TH1F* MyModule::testEtaHistogram = new TH1F("testMuonEta","Eta",40,&(*etabins));
TH1F* MyModule::testPtHistogram  = new TH1F("testMuonEta","Eta",150,&(*ptbins));
const Double_t MyModule::etabins[41] = {-2.0, -1.9, -1.8, -1.7, -1.6, -1.5, -1.4, -1.3, -1.2, -1.1,
                                         -1.0, -0.9, -0.8, -0.7, -0.6, -0.5, -0.4, -0.3, -0.2, -0.1,
                                         0.0,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,
                                         1.0,  1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9, 2.0};
const Double_t MyModule::ptbins[151]   = {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0,
                                          10.0,11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,
                                          20.0,21.0,22.0,23.0,24.0,25.0,26.0,27.0,28.0,29.0,
                                          30.0,31.0,32.0,33.0,34.0,35.0,36.0,37.0,38.0,39.0,
                                          40.0,41.0,42.0,43.0,44.0,45.0,46.0,47.0,48.0,49.0,
                                          50.0,51.0,52.0,53.0,54.0,55.0,56.0,57.0,58.0,59.0,
                                          60.0,61.0,62.0,63.0,64.0,65.0,66.0,67.0,68.0,69.0,
                                          70.0,71.0,72.0,73.0,74.0,75.0,76.0,77.0,78.0,79.0,
                                          80.0,81.0,82.0,83.0,84.0,85.0,86.0,87.0,88.0,89.0,
                                          90.0,91.0,92.0,93.0,94.0,95.0,96.0,97.0,98.0,99.0,
                                          100.0,101.0,102.0,103.0,104.0,105.0,106.0,107.0,108.0,109.0,
                                          110.0,111.0,112.0,113.0,114.0,115.0,116.0,117.0,118.0,119.0,
                                          120.0,121.0,122.0,123.0,124.0,125.0,126.0,127.0,128.0,129.0,
                                          130.0,131.0,132.0,133.0,134.0,135.0,136.0,137.0,138.0,139.0,
                                          140.0,141.0,142.0,143.0,144.0,145.0,146.0,147.0,148.0,149.0,150.0};
```

Notice that the number of bins the histograms are told to have (the 3rd argument) are 40 and 150, while the vectors defining what those bins are (the 4th argument) have 41 and 151 elements, respectively. Don't think in terms of counting the number of elements of the vector; think in terms of counting the number of "gaps" between the numbers of the vectors. These gaps are the bins to be filled by the η and P_T values.

Declared objects contain junk until they're assigned a value, so initialize the `TFile` object in the constructor with

```
_file = 0;
```

We want only one `TFile` object to be declared when we run some set of data through our module, so we instantiate it in `MyModule::BeginJob` with

```
_file  = new TFile("testrootfile.root", "RECREATE");
```

Next, let's go down to `MyModule::EndJob`, which executes after all the data is finished running through out module, and save the filled histograms to file.

```
int MyModule::EndJob()
{
   std::cout << "MyModule::EndJob" << std::endl;
   // Write histograms to the output file
   _file->cd();
   testEtaHistogram->Write();
   testPtHistogram->Write();
   _file->Close();

   return 0;
}
```

This is a good place to pause to recompile and run, even though the histograms are not being filled with anything yet. Be sure to do both. Sometimes syntax errors made with static members do not show up until run time. Also, check your directory to see if the `testrootfile.root` file appeared after running.

Moving on to actually filling these histograms, we will require a lorentz vector to calculate for us the quantities we are actually interested in, so add the following to the included files at the top of the source code:

```
#include "TLorentzVector.h"
```

Recall from the previous section where we learned how to fill and access the muon data block, that we wrote a `for` loop over over the muons for each event. Within that loop, write:

```
//Assign values for current reconstructed muon
const TLorentzVector* q = muo->Momentum();
Double_t reconeta = q->PseudoRapidity(); //(jmn)
Double_t reconet  = q->E();
Double_t reconphi = q->Phi();
Double_t reconpt = muo->TrackPt();

// Fill histogram
testEtaHistogram->Fill(reconeta);
testPtHistogram->Fill(reconpt);
```

This will create a lorentz vector for the muon, then return that values we are interested in filling out histograms with. I included a couple other values people commonly find useful as well.

We are now ready to recompile and run our module. After running 10000 events (remember to remove `cout` print statements!), I get the histograms in figures 1 and 2.

Create a Basic Ntuple

An ntuple is an object that stores information for each event passed through a module and can subsequently be access from `ROOT` to make histograms. As opposed to making histograms directly within our module as in the previous section "Make and Fill a Histogram," an ntuple allows far greater flexibility. If we wanted to change some aspect of the in-module-histogram (such as changing or adding a new cut), we would have to change the code, recompile, and rerun all the data. This can get extremely time consuming if we're running a large body of data. With an ntuple, however, we need only run the data once and then afterward play around with various cuts at the `ROOT` command prompt until the desired selection is found. We will try some explicit examples at the end of this section.

To accomplish this task, we will make another module containing our personal ntuple class. This class will be instantiated in `MyModule`, where we will also declare a `TTree` object [4]. This `TTree` object, for our present purposes, will have just a single branch (`TBranch` object [5]) into which our ntuple will be loaded. Finally, this ntuple-containing tree object will be saved to the same `TFile` that our simple histograms were in the previous section. We will again open this file with `ROOT` to access our ntuple information.

Starting from the top directory of our release, create and open the following two files:

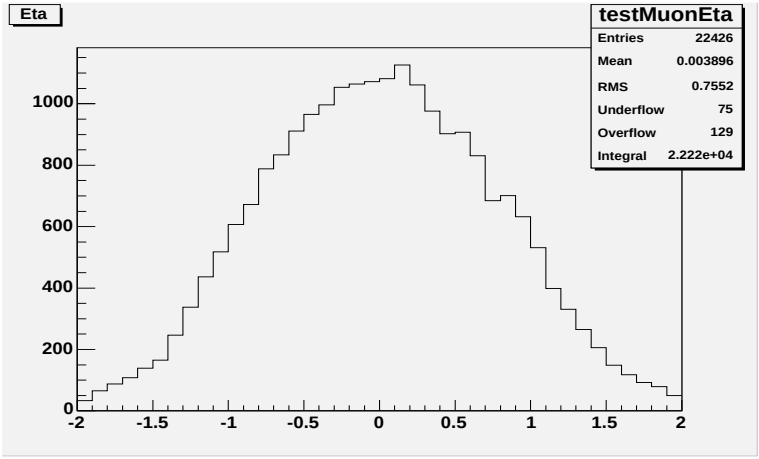

FIG. 1: η Distribution of muons for 10000 $WH \rightarrow l\nu b\bar{b}$ events.

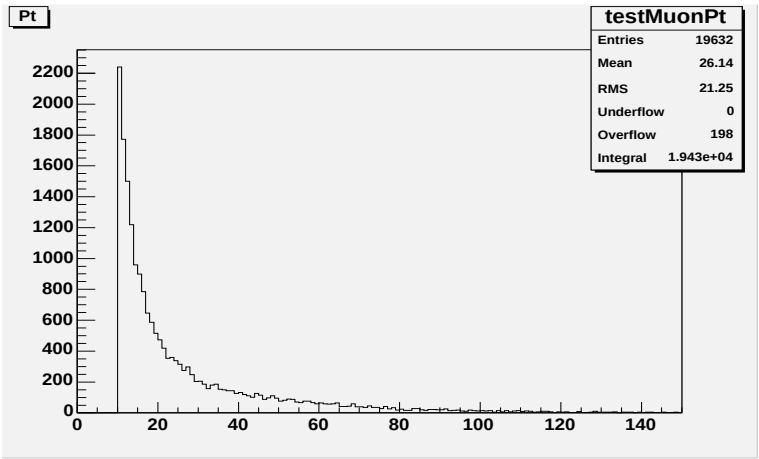

FIG. 2: P_T Distribution of muons for 10000 $WH \rightarrow l\nu b\bar{b}$ events.

```
xemacs MyAnalysis/src/MyNtuple.cc &
xemacs MyAnalysis/MyAnalysis/MyNtuple.hh &
```

Our ntuple source code will be mostly empty for now. Put into the source code file:

```
#include <math.h>
#include "MyAnalysis/MyNtuple.hh"

ClassImp(MyNtuple)

// Constructor
MyNtuple::MyNtuple()
{

}
// Destructor
MyNtuple::~MyNtuple()
{

}
```

Put into the header file:

```
#ifndef __MyNtuple__
#define __MyNtuple__
#include <iostream>
```

```
#include "TObject.h"
#include "TObjArray.h"

using namespace std;

class MyNtuple : public TObject
{
  public:
   // Class Constructor.
   MyNtuple();
   // Class destructor.
   ~MyNtuple();

  private:

   // Use this class in ROOT
   ClassDef(MyNtuple,1);
};

#endif
```

Of course, none of this will compile if we forget to make the `MyAnalysis/dict/MyNtuple_linkdef.h` file containing:

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

#pragma link C++ nestedclasses;
#pragma link C++ nestedtypedefs;

#pragma link C++ class MyNtuple;

#endif
```

This is a good place to pause and make sure what we have so far compiles and runs.

```
gmake MyAnalysis.nobin USESHLIBS=1
root -l -q ''MyAnalysis/drivers/myDriver(10,10)'' >& log
```

Now we need `MyNtuple` to communicate with `MyModule`. In `MyModule.hh`, write `#include ''MyAnalysis/MyNtuple.hh''` where the other `#include`'ed files are, and in the class declarations write:

```
class MyNtuple;
class TTree;
class TBranch;
```

In `MyModule.cc`, write `#include ''MyAnalysis/MyNtuple.hh''` at the top as well. Back to `MyModule.hh`, include the following lines under the `protected:` section of the class declaration:

```
TTree*   _tree;
TBranch* _branch;
MyNtuple* _ntuple;
```

That's it for the basic structure. Assuming that it still compiles and runs, let's move on to putting some content in our new ntuple. As a first example, lets start with some information about the muons. In `MyNtuple.hh` under the `public:` section, write in the following value-assignment functions:

```
// Fill Basic Info
void SetMuonEta      (Double_t   val) { _muonEta       = val; }
void SetMuonEt       (Double_t   val) { _muonEt        = val; }
void SetMuonPhi      (Double_t   val) { _muonPhi       = val; }
void SetMuonPt       (Double_t   val) { _muonPt        = val; }
```

In the same file under the `public:` member section again, declare the variables associated with the functions we just put in:

```
// Basic info
Double_t   _muonEta;
Double_t   _muonEt;
Double_t   _muonPhi;
Double_t   _muonPt;
```

And in the `private` section, include:

```
TTree*   _tree;
TBranch* _branch;
MyNtuple* _ntuple;
```

Now let's migrate to `MyModule.cc` and start making use of this new ntuple class we've created. Like usual, we start by initializing our declared objects in the constructor. Have your constructor now look like:

```
//Constructor
MyModule::MyModule():TStnModule("MyModule","MyModule")
{
  _file=0;
  _tree=0;
  _ntuple = new MyNtuple();

  // The first argument "tree" is the object called in ROOT to
  // make histograms.
  _tree = new TTree("tree","WH Background Ntuple");

  // Option B at http://root.cern.ch/root/html/TTree.html
  _branch = _tree->Branch("onebranch","MyNtuple",&_ntuple, 32000,1);
}
```

In `MyModule::EndJob`, write the tree object to file

```
_tree-> Write();
```

in the same place we wrote the histograms to file, after `_file->cd();` and before `_file->Close();`.

After compiling and running again, observe how this has affected our output file so far. Open the root file in the top directory of the release

```
root testrootfile.root
```

At the `ROOT` command prompt open a file browser:

```
TBrowser b
```

Double click on `ROOT files`; double click on `testrootfile.root`. Now, in addition to the histograms from the previous section, we see a folder for the `TTree` object we made, inwhich we find a single "branch" called `onebranch`, inwhich we find "leaves" for each variable declared in our ntuple. Since we have not assigned values to these variables, these leaves are still empty.

We are just about ready to move on to `MyModule::Event` to start filling our ntuple, but we must make one digression first. It is important to have a `Clear()` function in our ntuple. Sometimes a particular value of our ntuple may not be updated for some particular event, in which case the value from the previous event would be counted (or double-counted, actually). Hence, we want to create a simple function that will reset all variables to some default value. This default value should be nonphysical; it should be a value that simply will not occur as a legitimate experimental value. For instance, assign any counters the value of -1. In `MyNtuple.hh`, declare the new function

```
void Clear();
```

under the `public:` section. In `MyNtuple.cc`, write in the following new function body:

```
void MyNtuple::Clear()
{
   // Be sure to reset the values to physically
   // unrealistic numbers.
    _muonEta = -999.0;
    _muonEt = -1;
    _muonPhi = -999.0;
    _muonPt = -1;
}
```

At last, let's start filling our ntuple with information. We should still have a `for` loop over the muons of an event in `MyModule::Event` from a previous section of this document. If not, edit it so that it now looks like:

```
int MyModule::Event(int ientry)
{
   std::cout << "MyModule::Event" << std::endl;

   // Clear the ntuple
   _ntuple->Clear();

   // Fill Stntuple Data Blocks
   FillDataBlocks(ientry);

   // Access the number of muons recorded for this event.
   int nmuons=fMuonBlock->NMuons();//returns the number of muons for this event
   std::cout << "  nmuons = " << nmuons << std::endl;
   for(int imu=0; imu<nmuons; imu++) //loop over the reconstructed muons
     {
```

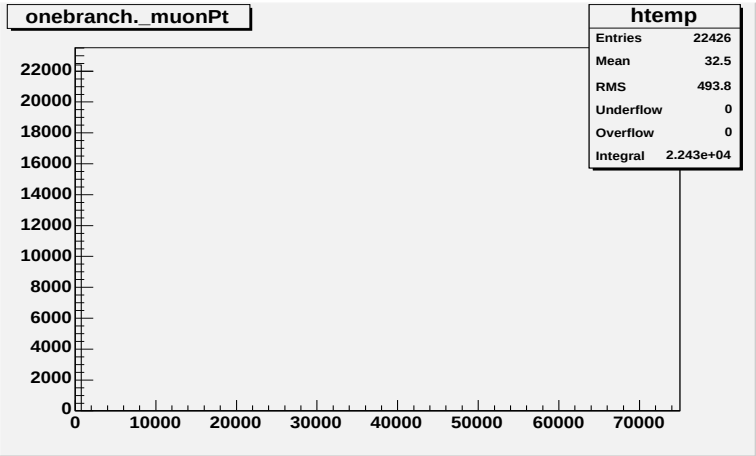

FIG. 3: `tree->Draw(''_muonPt'')`

```
        TStnMuon* muon =  fMuonBlock->Muon(imu);//fMuonBlock is a TStnMuonBlock.

        //Assign values for current reconstructed muon
        const TLorentzVector* q = muon->Momentum();
        _ntuple->SetMuonEta(q->PseudoRapidity());
        _ntuple->SetMuonEt (q->E());
        _ntuple->SetMuonPhi(q->Phi());
        _ntuple->SetMuonPt (q->Pt());
        std::cout << " pt = " << q->Pt() << std::endl;

        _tree->Fill();
      }
    return 0;
}
```

First, note that we cleared the ntuple information at the beginning of the event. If, for intance, an event had no muons, then the variables containing muon information would not acquire new values in the loop. This may lead to wrong information being inadvertantly filled if we are not careful. Second, notice that the tree object calls its `Fill` function inside the loop. In this case, there may be several muons in a single event. If we filled the tree after the loop, we would only be filling the information of the last muon in the list for a particular event.

Let's compile, run, and look in the `ROOT` file again. When I run 10000 events and then double click on the leaf titled `_muonPt`, I get the rather useless plot in figure 3. I can get the same plot at the `ROOT` command line by typing

```
tree->Draw(''_muonPt'')
```

This is where we illustrate the power of using an ntuple over making in-module histograms. We can see from the axes of figure 3 that there must be a few mismeasured muons at extremely high P_T. So let's consider only muons with $P_T < 200$ GeV. Also, it is often the case that we only want to consider muons above some minimum P_T threshold. So let's also restrict ourselves to muons with $P_T > 10$ GeV. The syntax at the `ROOT` command line is

```
tree->Draw(''_muonPt'',''_muonPt<200 && _muonPt>10'')
```

This yields figure 4.

We can also make combinations of cuts between different kinds of values. Suppose we decide we want to look at the pseudorapidity distribution of muons. We would type

```
tree->Draw(''_muonEta'')
```

and get figure 5. However, we then realize we don't want the junk muons (very low and very high P_T muons), just as before. So we include some additional cuts

```
tree->Draw(''_muonEta'',''_muonPt<200 && _muonPt>10'')
```

and get figure 6. Again, throughout all this manipulation of cuts to get exactly the combinations we want, we never had to rerun the data.

Create a Better Ntuple For A More Complex Analysis

The previous section provides a good introduction to what an ntuple is and does, but is only adequate for the most basic of analyses. The ntuple in the previous section only saved numerical values; a more general use of ntuples allows for more complex instances of classes to be saved as well.

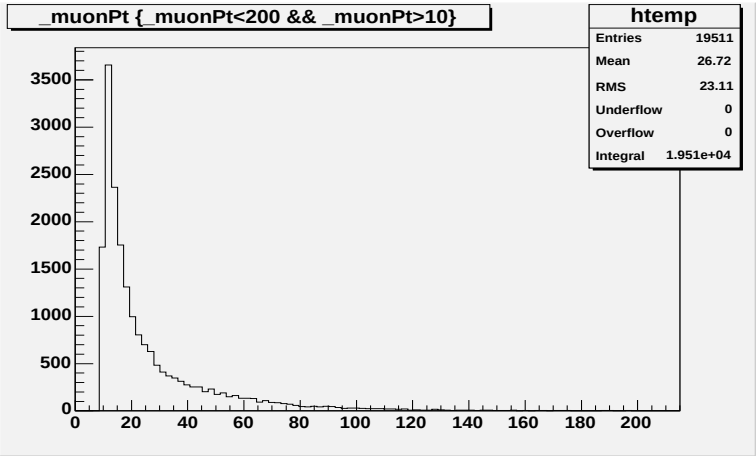

FIG. 4: `tree->Draw(''_muonPt'',''_muonPt<200 && _muonPt>10'')`

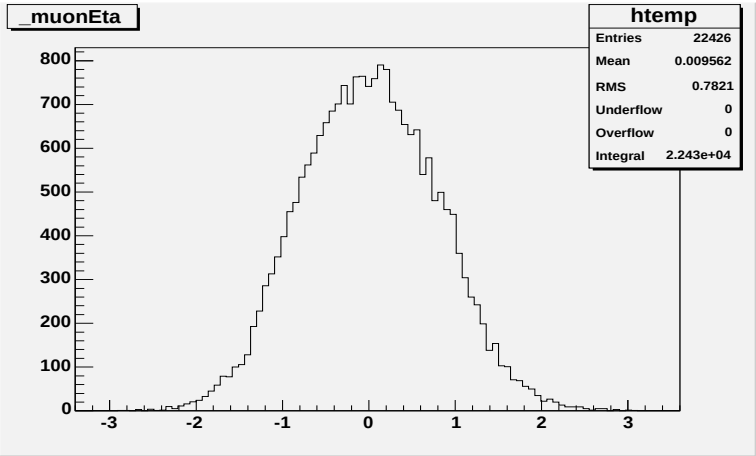

FIG. 5: `tree->Draw(''_muonEta'')`

For instance, any particular event has a single-valued header variables (TStnHeaderBlock) that can be stored to the ntuple, whereas the same event may have several muons with several different values to be stored for each event. It is then a convenient and common practice to create a new class for these muons that records all relevant information for a particular muon, then make an array of instances of this muon class that is stored in the ntuple.

It was mentioned in the previous section that one of the reasons for having ntuples is to store event data in a smaller, more managable form since we are hand-picking what information we want. There are a myriad of ways to utilize ntuples in an analysis, but in this section we will expand upon the previous one in the following particular manner. Instead of saving single variables to the ntuple we will save more complex C++ objects. The function of `MyModule` here will be just to pick and choose what information we want to save to our ntuple from each event. We will subsequently create a new module `MyAnalyzer` whose purpose will be to retrieve information from the ntuple and calculate desired quantities from the stored information. Those desired quantities will then be stored into a simple new data structure in a new `ROOT` file that we can access in a manner similar to the previous section.

The reasoning behind this strategy is that we should hopefully be able to run `MyModule` once over a large amount of data. This may take many hours. Once our ntuple is filled, we can perform whatever calculations are desired in `MyAnalyzer`. Since it is expected that we will need to continuously play with and alter these calculations, it is imperative that we not be required to wait those long hours every time we wish to alter some calculation. Thus, it will be `MyModule`'s job to fill the ntuple with the "raw" data from the Stntuple, load it into a `TTree`, and save the tree to a `ROOT` file. `MyAnalyzer`'s purpose will be to calculate desired quantities from the data stored in that `ROOT` file.

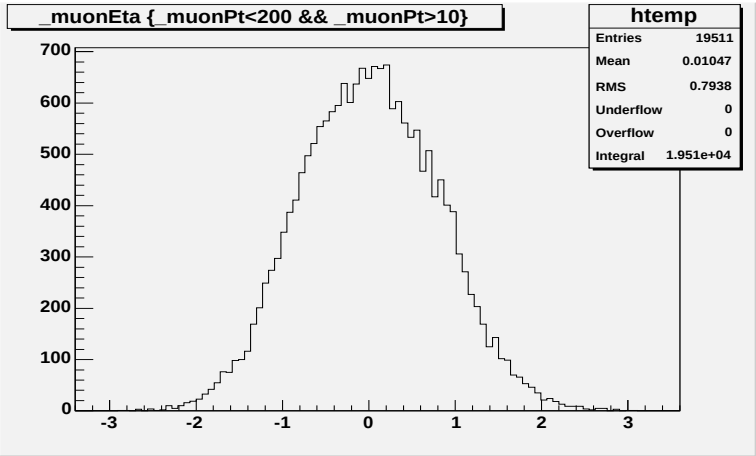

FIG. 6: `tree->Draw(''_muonEta'',''_muonPt<200 && _muonPt>10'')`

Storing an Object to an Ntuple

For our first task, let's create a new class that stores information from `TStnHeaderBlock`. These variables will only need to be filled once per event so this will be mostly a pedagogical task concerning how to add more complex objects to our ntuple. As always, the first and most easily forgotten step is to make our `*linkdef.h` file so that the new module will compile. In the folder `stn_example/MyAnalysis/dict` create a file called `MyHeaderBlock_linkdef.h`. Write in this file:

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

#pragma link C++ nestedclasses;
#pragma link C++ nestedtypedefs;

#pragma link C++ class MyHeaderBlock;

#endif
```

Now create a source code file (`stn_example/MyAnalysis/src/MyHeaderBlock.cc`) and a header file (`stn_example/MyAnalysis/MyAnalysis/MyHeaderBlock.hh`).

Put in the source code file:

```
#include <math.h>
#include <iostream>
#include "Stntuple/obj/TStnHeaderBlock.hh"
#include "TLorentzVector.h"
using namespace std;
#include "MyAnalysis/MyHeaderBlock.hh"

ClassImp(MyHeaderBlock)

MyHeaderBlock::MyHeaderBlock() : TObject()
{
  _eventNumber     = -1;
  _runNumber       = -1;
  _sectionNumber   = -1;
  _instLum         = -1.0;
}

MyHeaderBlock::~MyHeaderBlock(){}

void MyHeaderBlock::Clear()
{
  _eventNumber     = -1;
  _runNumber       = -1;
  _sectionNumber   = -1;
  _instLum         = -1.0;
}

void MyHeaderBlock::Print()
{
```

```
  std::cout << "----------------------------------------"   << std::endl;
  std::cout << "---------- My Header Block ------------"   << std::endl;
  std::cout << "----------------------------------------"   << std::endl;
  std::cout << "-- Event Number = "      << _eventNumber   << std::endl;
  std::cout << "-- Run Number = "        << _runNumber     << std::endl;
  std::cout << "-- Section Number = "    << _sectionNumber << std::endl;
  std::cout << "-- InstLum = "           << _instLum       << std::endl;
  std::cout << std::endl;
}
```

In the header file, put:

```
#ifndef _MYHEADERBLOCK_HH_
#define _MYHEADERBLOCK_HH_

#include "TObject.h"
#include "TLorentzVector.h"
#include "Stntuple/obj/TStnHeaderBlock.hh"

class TStnHeaderBlock;

class MyHeaderBlock : public TObject
{
public:

  MyHeaderBlock();
  ~MyHeaderBlock();

  // Return event info
  Int_t    EventNumber    () { return _eventNumber;   }
  Int_t    RunNumber      () { return _runNumber;     }
  Int_t    SectionNumber  () { return _sectionNumber; }
  Float_t  InstLum        () { return _instLum;       }

  // Filling muon info
  void SetEventNumber    (Int_t     val) { _eventNumber   = val; }
  void SetRunNumber      (Int_t     val) { _runNumber     = val; }
  void SetSectionNumber  (Int_t     val) { _sectionNumber = val; }
  void SetInstLum        (Float_t   val) { _instLum       = val; }

  // Clear contents
  void Clear();
  void Print();

protected:

  Int_t         _eventNumber;
  Int_t         _runNumber;
  Int_t         _sectionNumber;
  Float_t       _instLum;

  ClassDef(MyHeaderBlock,1);
};

#endif
```

Assuming `MyHeaderBlock` now compiles and runs properly, let's move on to altering our ntuple (`MyNtuple`) so that it will accept and store such objects. Remove any variable declarations and functions related to filling or returning those variables– we're rewriting this (if the `Clear()` function is still there, leave it, but empty). Don't forget to remove the same variables from the source code file as well. Lastly, go to `MyModule::Event()` and remove any lines that call the functions we just deleted. We're restarting construction of the ntuple module from scratch, so just leave the skeleton structure.

Now the source file for `MyNtuple` should be

```
#include <math.h>
#include "MyAnalysis/MyNtuple.hh"
#include "MyAnalysis/MyHeaderBlock.hh"

ClassImp(MyNtuple)

//Constructor
MyNtuple::MyNtuple()
{
  _headerBlock   = new MyHeaderBlock();
}
//Destructor
MyNtuple::~MyNtuple(){}

void MyNtuple::Clear()
{
  _headerBlock->Clear();
}
```

and the header file for `MyNtuple` should be

```
#ifndef __MyNtuple__
#define __MyNtuple__

#include <iostream>

#include "TObject.h"
#include "TObjArray.h"
#include "MyAnalysis/MyHeaderBlock.hh"

using namespace std;

class MyNtuple : public TObject
{
  public:
   // Class Constructor.
   MyNtuple();
   // Class destructor.
   ~MyNtuple();

   // Return objects from a filled ntuple
   MyHeaderBlock*   NtHeaderBlock   () { return _headerBlock; }

   // Add objects to our ntuple
   void AddHeaderBlock     (MyHeaderBlock*    block) { _headerBlock  = block; }

   // Object Declarations
   // (These members must be public for the analyzer module to access them)
   MyHeaderBlock*    _headerBlock;

   void Clear();

 private:

  // Use this class in ROOT
   ClassDef(MyNtuple,1);
};
#endif
```

Now we are ready to fill this in `MyModule`. Include our new header block in both `MyModule.hh` and `MyModule.cc` with

```
#include "MyAnalysis/MyHeaderBlock.hh"
```

Change `MyModule::Event` so that it looks like:

```
int MyModule::Event(int ientry)
{
   //std::cout << "MyModule::Event" << std::endl;

   // Clear the ntuple
   _ntuple->Clear();

   // Fill Stntuple Data Blocks
   FillDataBlocks(ientry);

   // Fill our header block and add to the ntuple
   MyHeaderBlock* header = new MyHeaderBlock();
   header->SetEventNumber  (GetHeaderBlock()->EventNumber());
   header->SetRunNumber    (GetHeaderBlock()->RunNumber());
   header->SetSectionNumber(GetHeaderBlock()->SectionNumber());
   header->SetInstLum      (GetHeaderBlock()->InstLum());
   _ntuple->AddHeaderBlock(header);

   // Fill the values assigned to the tree (our ntuple) with
   // the values for this event.
   _tree->Fill();

   return 0;
}
```

If we again enter the `ROOT` file output, open a browser, and go into our tree we will find a leaf for our header object. However, we will not be able to access the data members the header object contains anymore. This will be put off until the output of the analyzer module we have yet to make.

Storing an Array of Objects in an Ntuple

Before putting our analyzer module together, let's see how to store an array of objects to our ntuple. Whereas the header information is unique for each event, there may be several muons per event that each have information to be recorded. As such, we will construct a `MyMuon` class in a manner completely analogous to `MyHeaderBlock`. Then, instead of saving instances of `MyMuon` to the ntuple, we will make an array (`TObjArray`) of muon objects within `MyModule` and save that to the ntuple. This way, we are still consistently filling the tree once per event.

Following what we did for `MyHeaderBlock`, begin by making the `stn_example/MyAnalysis/dict/MyMuon_linkdef.h` file as before. Then make a header file (`MyAnalysis/MyAnalysis/MyMuon.hh`) containing:

```
#ifndef _MYMUON_HH_
#define _MYMUON_HH_
#include "TObject.h"
#include "TLorentzVector.h"
#include "Stntuple/obj/TStnMuon.hh"
#include "Stntuple/obj/TStnTrack.hh"

class TStnMuon;

class MyMuon : public TObject
{
public:

  MyMuon();
  ~MyMuon();

  // Return event info
  Double_t     Eta          () { return _eta;  }
  Double_t     Et           () { return _et;   }
  Double_t     Phi          () { return _phi;  }
  Double_t     Pt           () { return _pt;   }

  // Filling muon info
  void SetEta          (Double_t   val) { _eta      = val; }
  void SetEt           (Double_t   val) { _et       = val; }
  void SetPhi          (Double_t   val) { _phi      = val; }
  void SetPt           (Double_t   val) { _pt       = val; }

  // Clear Contents
  void Clear();

protected:

  // Basic muon info
  Double_t   _eta;
  Double_t   _et;
  Double_t   _phi;
  Double_t   _pt;

  ClassDef(MyMuon,1);
};

#endif
```

And make the source code (`MyAnalysis/src/MyMuon.cc`) containing:

```
#include "TLorentzVector.h"
using namespace std;
#include "MyAnalysis/MyMuon.hh"

ClassImp(MyMuon)

   MyMuon::MyMuon() : TObject()
{
   _eta        = -999.0;
   _et         = -999.0;
   _phi        = -999.0;
   _pt         = -999.0;
}

MyMuon::~MyMuon(){}

void MyMuon::Clear()
{
   _eta        = -999.0;
   _et         = -999.0;
   _phi        = -999.0;
   _pt         = -999.0;
}
```

Let's go back to `MyNtuple` and let it accept `TObjArray` objects. It is important to note that `MyMuon` inherits from `TObject`, which is what allows us to load `MyMuon` instances into a `TObjArray` object.

First, check `MyNtuple` to see if `TObjArray`'s are included:

```
#include "TObjArray.h"
```

Then write in an object declaration;

```
TObjArray*   _muonArray;
```

then a function that adds a `TObjArray` to the ntuple;

```
void AddMuonList  (TObjArray*   list ) { _muonArray  = list; }
```

and finally a function that will return the `TObjArray`

```
TObjArray*  NtMuonList() { return _muonArray;  }
```

Moving on to `MyNtuple.cc`, write the following declaration into the constructor:

```
_muonArray  = new TObjArray();
_muonArray->SetOwner();
```

The purpose of `SetOwner()` is to "Set whether this collection is the owner (enable==true) of its content. If it is the owner of its contents, these objects will be deleted whenever the collection itself is deleted." [6] In the `Clear()` function, write in the line

```
_muonArray->Clear();
```

We are finally ready to declare and fill `MyMuon` objects in `MyModule` and then form them into an array that is saved to the ntuple. Let's begin with `MyModule.hh`. We first need to include the our new muon class:

```
#include ''MyAnalysis/MyMuon.hh''
```

Write in the line

```
class MyMuon;
```

in the appropriate place, and declare the object array under the protected class members.

```
TObjArray*  _muonArray;
```

In the source code `MyModule.cc`, again include our muon class:

```
#include ''MyAnalysis/MyMuon.hh''
```

Instantiate the `TObjArray` that was declared in the header in the constructor:

```
_muonArray = new TObjArray();
```

The last step in filling muon informtation to our ntuple is to put the following lines in `MyModule::Event()` just after (or before, it doesn't matter) the header values we put in just above:

```
// Fill the array of MyMuon objects
int nmuons=fMuonBlock->NMuons();//returns fNMuons from TStnMuonBlock.hh
for(int imu=0; imu<nmuons; imu++) //loop over the reconstructed muons
  {
    //Access reconstructed muons information
    TStnMuon* muon =  fMuonBlock->Muon(imu);//fMuonBlock is a TStnMuonBlock.
    TStnTrack* trk =  fTrackBlock->Track(muon->TrackNumber());
    const TLorentzVector* q = muon->Momentum();
    MyMuon* mymuon = new MyMuon();
    mymuon->SetEta           (q->PseudoRapidity());
    mymuon->SetEt            (q->E());
    mymuon->SetPhi           (q->Phi());
    mymuon->SetPt            (q->Pt());
    _muonArray->Add(mymuon);
  }
_ntuple->AddMuonList(_muonArray);
```

Building an Analyzer Module for an Ntuple

At last, we are ready to build an analyzer module. This new module will take the output of `MyModule` (`testrootfile.root`) as it's input, access the `TTree` object that contains our now-filled ntuple, perform desired calculations on these values, then save these new desired values to a new `TTree` which will be saved in a new `ROOT` file. In this particular example, our ntuple has a very limited amount of information saved in it so I will not actually calculate new values. Instead, I'll just show how to access some of the data in the ntuple and resave the same values to the aforementioned data structure. This may seem to be a redundant step (and it is) since we're just resaving the same numbers, but there are several pedagogical reasons for doing so. First, this section is already far more expansive than the first section on creating ntuples and it doesn't need any further unnecessary complexity. Second, we have already covered saving information into a `TTree`; retrieving information from a `TTree` is no less trivial and the central point at present. Third, any particular calculations would detract from the generality intended in this document. The kinds of calculations that could be found in this kind of analyzer module would be the application of specific trigger requirements, calculating the invariant mass of jets, etc.

As always, every new module requires a `*linkdef.h` file. So create `stn_example/dict/MyAnalyzer_linkdef.h` containing:

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

#pragma link C++ nestedclasses;
#pragma link C++ nestedtypedefs;

#pragma link C++ class MyAnalyzer;

#endif
```

Since this analyzer module will be run separately from `MyModule`, we will need a new driver script that loads `testrootfile.root`. Create the file `/stn_example/MyAnalysis/drivers/AnalyzerDriver.C` and put in it:

```
#include <iostream>
TChain*   treeChain;
MyAnalyzer* analyzer;

void AnalyzerDriver(int nevents = 10)
{
  std::cout << "Begin analyzerDriver" << std::endl;

  // The argument must have the name of the TTree instance
  // save to the TFile.
  treeChain = new TChain("tree");

  // Of course, "jnett" is my home directory, you'll have
  // to change that.
  treeChain->Add("~jnett80/stn_example/testrootfile.root");
  analyzer = new MyAnalyzer();
  analyzer->Run(treeChain,nevents);

  std::cout << "End analyzerDriver" << std::endl;
}
```

Now declare our analyzer class in a header file `stn_example/MyAnalysis/MyAnalysis/MyAnalyzer.hh`

```
#ifndef _MYANALYZER_HH_
#define _MYANALYZER_HH_

#include <sstream>
#include <string>

#include <iostream>
#include <iomanip>
#include <ostream>
#include <fstream>

#include "TObject.h"
#include "TLorentzVector.h"
#include "TMath.h"
#include "TTree.h"
#include "TChain.h"
#include "TH1D.h"
#include "TFile.h"

#include "MyAnalysis/MyNtuple.hh"

using namespace std;

class TFile;
class TTree;
class MyNtuple;

class MyAnalyzer : public TObject
{
public:
  MyAnalyzer();
  ~MyAnalyzer();

  // Declare my output structure.  The values
  // that will be put into this are the ones we want
  // in our final TTree object.  For this module, instead
  // of saving an ntuple object to a TTree as we did
  // with MyModule, we will save this struct to a TTree.
  struct MyEventInfo
  {

  };

  // This string will be used to declare the
  // variables of the above struct in a TBranch
  // of the output TTree object.
  static const TString eventInfoDeclare;
```

```
  void Run(TChain* treeChain, int nevents = 10);

protected:

  // These file, tree, and branch objects will be the
  // new output just like we have in MyModule
  TFile*             _processedFile;
  TTree*             _processedTree;
  TBranch*           _processedBranch;

  // Instantiate my output structure
  MyEventInfo eventInfo;

  // Analysis functions
  void EndAnalyzer();

  ClassDef(MyAnalyzer,1)
};

#endif
```

For the source code of our analyzer module (`stn_example/MyAnalysis/src/MyAnalyzer.cc`), begin with

```
#include <iostream>
#include "MyAnalysis/MyAnalyzer.hh"
#include "MyAnalysis/MyHeaderBlock.hh"
#include "MyAnalysis/MyMuon.hh"
#include "MyAnalysis/MyNtuple.hh"
#include "TObject.h"
#include "TObjArray.h"
#include "TLeaf.h"

ClassImp(MyAnalyzer)

  using namespace std;

// To be filled with the names of the variables
// that will be in the output.  The format follows that
// given in ''Case A'' on http://root.cern.ch/root/html/TTree.html.
const TString MyAnalyzer::eventInfoDeclare = "";

// Constructor
MyAnalyzer::MyAnalyzer() : TObject()
{
  // Make a new TTree for the new results.  The new ROOT file
  // containing the output is "testProcessedFile.root
  _processedTree = 0;
  _processedFile = 0;
  _processedTree = new TTree("ProcessedTree","My Processed Ntuple");
  _processedFile = new TFile("testProcessedFile.root","RECREATE");
  _processedBranch = _processedTree->Branch("eventInfo",&eventInfo, eventInfoDeclare);
}

// Destructor
MyAnalyzer::~MyAnalyzer(){}

void MyAnalyzer::Run(TChain* treeChain, int maxEvents)
{
  std::cout << "  Begin MyAnalyzer::Run" << std::endl;
  EndAnalyzer();
  std::cout << "  End MyAnalyzer::Run" << std::endl;
}

void MyAnalyzer::EndAnalyzer()
{
  _processedFile->cd();
  _processedTree->Write();
  _processedFile->Close();
}
```

Run the analyzer module with

```
root -l -q ''MyAnalysis/drivers/AnalyzerDriver(10)''
```

Let's put some content in `MyAnalyzer`. In the header file, change the declared (but still empty) data structure so that it contains variables for the header info:

```
struct MyEventInfo
{
  Int_t     eventNumber;
  Int_t     runNumber;
  Int_t     sectionNumber;
  Float_t   instLum;
};
```

Along with this, we must change the static string in the source code from `const TString MyAnalyzer::eventInfoDeclare = ‘‘’’;` to

```
const TString MyAnalyzer::eventInfoDeclare =
  "eventNumber/I:"
  "runNumber/I:"
  "sectionNumber/I:"
  "instLum/F";
```

Now the output `TTree` contains a branch that has these variables reported. So far, they are still empty. We now move to `MyAnalyzer::Run` in the source code to see how to extract ntuple information from the output of `MyModule`.

In the following, we access the information in the filled-ntuple and then save it to the struct declared in the header of the analyzer module. The purpose of individual lines are commented in the code. So alter `MyAnalyzer::Run` to look like the following:

```
void MyAnalyzer::Run(TChain* treeChain, int maxEvents)
{
  std::cout << "  Begin MyAnalyzer::Run" << std::endl;

  // We don't want to accidentally try accessing more events than what
  // were saved to the ntuple in MyModule.  Doing so would likely
  // result in the dreaded segmentation error.
  int nevents = TMath::Min(int(treeChain->GetEntries()),maxEvents);

  // Create a new ntuple object whose address will be set to the
  // filled ntuple in the input TTree.
  MyNtuple* _ntuple = new MyNtuple();

  // Setting the address of _ntuple to the ntuple branch of the
  // input TTree.  Note that this will NOT fill _ntuple with any
  // of those values yet.
  treeChain->SetBranchAddress("branch",&_ntuple);

  // Recall that in MyModule we executed the tree->Fill() command
  // at the end of each event.  The following loop will cycle
  // back over each of these 'fills'.
  for(int ievent=0; ievent < nevents; ievent++)
    {
      // This is the command that fills _ntuple
      // with the values of a particular event saved
      // to the input TTree.
      int exists = treeChain->GetEntry(ievent);

      // Retrieve the header block that was saved to the ntuple in MyModule
      MyHeaderBlock* _header   = _ntuple->NtHeaderBlock();

      // Retrieve the values of the header block for this event.
      int _eventNumber   = _header->EventNumber();
      int _runNumber     = _header->RunNumber();
      int _sectionNumber = _header->SectionNumber();
      float _instLum     = _header->InstLum();

      // Fill these retrieved values into the struct "eventInfo".
      eventInfo.eventNumber      = _eventNumber;
      eventInfo.runNumber        = _runNumber;
      eventInfo.sectionNumber    = _sectionNumber;
      eventInfo.instLum          = _instLum;

      // Fill the output TTree
      _processedTree->Fill();
    }

  EndAnalyzer();
  std::cout << "  End MyAnalyzer::Run" << std::endl;
}
```

This should be a sufficient start to constructing an Stntuple-based analysis.

COMMON ERRORS

Changing Code within `Stntuple/loop`

If for any reason you need to change code within the header file of any module within `Stntuple/Stntuple/loop` you must, of course, compile with

```
gmake Stntuple._loop USESHLIBS=1
```

However, because of how intertwined these modules are with those in `Stntuple/Stntuple/ana` it is possible that you will need to compile that section of Stntuple as well, even if you haven't touched it's modules:

```
gmake Stntuple._ana USESHLIBS=1
```

For instance, I discovered this when changing the number of variables in a function of `TStnAna`. After compiling `TStnAna` and my own module I received the following error when running the module:

```
dlopen error: /mnt/autofs/misc/nuwm01.home/jnett80/stn_rel/shlib/Linux2_SL-GCC_3_4/
libStntuple_loop.so:
undefined symbol: _ZN10TStnModule5EventEiiRiS0_S0_S0_S0_S0_S0_S0_S0_S0_
R4TH2FR4TH1FS4_S4_S4_S4_S4_S4_S4_S4_S4_S4_S4_S4_S4_
S4_S4_S4_S4_S4_S4_S4_S4_S4_S4_S4_S4_S4_
Load Error: Failed to load Dynamic link library
/mnt/autofs/misc/nuwm01.home/jnett80/stn_rel/shlib/Linux2_SL-GCC_3_4/libStntuple_loop.so
*** Interpreter error recovered ***
root [0]
Processing TriggerAna/scripts/higgs.C(10,10)...
Error: Symbol TStnAna is not defined in current scope
FILE:TriggerAna/scripts/higgs.C LINE:3
Error: Symbol x is not defined in current scope
FILE:TriggerAna/scripts/higgs.C LINE:3
```

`dlopen error:` means that a shared library cannot be loaded into `ROOT`. In this case, `libStntuple_loop.so` cannot be loaded. This kind of error is then always (as far as I've seen) followed by the statement `Load Error: Failed to load Dynamic link library`. Notice that the one clue to the source of the problem is contained withing the "undefined symbol" `_ZN10TStnModule5EventEiiRiS0...`. I resolved this by first deleting all the libraries with `gmake clean` and then recompile everything, including now `gmake Stntuple._ana USESHLIBS=1`. Further changes of this sort can avoid the `gmake clean` bomb by compiling `libStntuple_ana.so` before attempting to run the module.

I've actually encountered this failure-to-load a shared library error for several reasons (another time was when I mixed up the syntax for the destructor in my `TStnModule` module). It is particularly bedeviling because it is not a compilation error; it occurs when `rootlogon.C` is attempting to load the necessary shared libraries for the ROOT driver script. The only clue you get is that there is a problem somewhere within the directory of the indicated shared library.

The missing separator error

An often encountered error contains the mystifying statement `missing separator. Stop.`. A frequent solution to this is to `gmake MyModule.clean` to completely clear the libraries, and then recompile normally `gmake MyModule.nobin USESHLIBS=1`.

---

[1] http://www.opengroup.org/onlinepubs/009695399/functions/unsetenv.html
[2] Thanks to Matteo Bauce for advice on the second case.
[3] http://www.cplusplus.com/doc/tutorial/classes2.html
[4] http://root.cern.ch/root/html/TTree.html
[5] http://root.cern.ch/root/html/TBranch.html
[6] http://root.cern.ch/root/html/TCollection.html#TCollection:SetOwner