User Guide to Common Analysis Format

Reiner Hauser, rhauser@fnal.gov v0.2, 2 February 2005

This document describes how to use the D0 Common Analysis Format (CAF).

Contents

1 Introduction

This document is a tutorial for the Common Analysis Format (CAF).

This ROOT based format consists of several D0 packages, `kinem_util`, `met_util` and `tmb_tree`. For historical reasons most classes live in the `tmb_tree` package and have a `TMB` prefix before the name, e.g. `TMBJet`.

2 Getting Started

In the following we assume that all examples are executed in a normal D0 work environment. However, except for SAM related things, everything should also work in a stand-alone environment as long as a ROOT installation is available. See the appendix for how to set up the software on a non-D0 system.

2.1 Setting up the Release [BEGINNER]

```
% setup D0RunII t05.02.00                    [1]
% newrel -t t05.02.00 work                   [2]
% cd work                                    [3]
% setenv LINK_SHARED yes                     [4]
% d0setwa                                    [5]
% set path = ( `pwd`/shbin/${SRT_SUBDIR} $path ) [6]
```

or, if you are using `bash` instead of `tcsh`:

```
% export LINK_SHARED=yes                     [4]
% export PATH=`pwd`/shbin/${SRT_SUBDIR}:${PATH} [6]
```

The steps should be familiar if you are used to the D0 software environment.

1. Set up the correct release version.
2. Create a new work area.
3. Enter the work area.
4. Enable shared libraries and binaries.
5. Set the current working area.
6. Add the shared binaries to your path.

2.2 One-time Preparations [BEGINNER]

In your work area you should execute the following command once:

```
% autoroot.py tmb_tree cafe
```

This will create a `.rootmap` file in your current directory. This is only needed when you run `root` interactively, to make your life easier. If `root` finds this file it will know which libraries to load for which classes, and you won't get any funny warnings when you open a CAF file.

2.3 Simple Examples [BEGINNER]

Note that in the following we assume that you execute ROOT from this directory. ROOT looks by default for certain configuration files in the current directory, especially `.rootmap`. If you move to a different directory, you have to either copy this file or move it to your home directory, where ROOT will also look for it.

In the following we assume you have an root file in CAF format named `input.root` available in your work directory. Just replace the file name with whatever you are preferring.

```
% root
```

This starts your root session. If you have done all the steps before, you should be able to access the CAF classes. Try it by typing:

```
root [0] TMBMuon m
root [1] m.Pt()
```

This will create and print an (uninitialized) TMBMuon object. There is no need to load the shared libraries explicitly if you've followed the previous recipe. Now let's open the file:

```
root [2] TFile file("input.root");
root [3] TTree *tree = (TTree *)file.Get("TMBTree");
```

You can now e.g. draw the variables in the CAF tree or use the `TBrowser` to view them.

```
root [4] tree->Draw("Muon.Pt()", "Muon.Pt() < 40.0")
root [5] tree->Draw("Muon.Pt():Muon.Eta()", "Muon.Pt() < 40.0")
root [6] new TBrowser()
```

2.4 Explicit Loading of the CAF Classes [ADVANCED]

If you want to compile code on the fly with ROOT's ACliC system, you need to load the necessary libraries explicitly, in addition to setting the include paths:

```
root [0] gSystem->Load("libPhysics.so")
root [1] gSystem->Load("libmet_util.so")
root [2] gSystem->Load("libtmb_tree.so")
root [3] gSystem->Load("libcafe.so")
root [4] .include $SRT_PRIVATE_CONTEXT/include
root [4] .include $SRT_PUBLIC_CONTEXT/include
```

Afterwards all CAF classes should be available in the interactive session. If you have used the `autoroot.py` script, you can also trigger the loading of all necessary libraries by a statement like:

```
root [0] TMBMuon m;
```

3 Using the CAF Environment (Cafe)

The CAF environment is a set of classes which make it easier to work with the objects in the common analysis format. More specifically, it includes an `Event` class that integrates all the various branches and provides type safe access to all the physics objects. In addition it has framework classes which make it easy to break your analysis into little packages and run them in any given order, share code with others etc.

You can use many of the CAFE classes in an interactive session as well, which is the way we introduce most of them here.

3.1 The `cafe::Event` Class [BEGINNER]

The `cafe::Event` class provides easy access to all common physics objects used in an analysis. Here is one way to use it:

```
root [0] cafe::Event event;
root [1] TFile file("input.root");
root [2] TTree *tree = (TTree *)file.Get("TMBTree");
root [3] event.setBranchAddresses(tree);
```

Now every call to `tree->GetEntry(i)` will fill the `event` object.

In practice, you only have to call `tree->LoadTree(i)`. This avoids that the full event is read at once. Instead the `Event` class will read the data as you ask for it, which can speed up processing tremendously.

3.2 Using the `cafe::Event` Class [BEGINNER]

The interface of this class provides methods to access all common physics objects. Here is how the interface looks like:

```
class Event : public EventBase {
public:
    Event();

    Collection<TMBMuon>      getMuons()    const;
    Collection<TMBEMCluster> getEMcellNN()  const;
    Collection<TMBEMCluster> getEMscone()  const;
    Collection<TMBJet>       getJCCA()     const;
    Collection<TMBJet>       getJCCB()     const;
    Collection<TMBJet>       getJets(const std::string& name) const;
    Collection<TMBTrack>     getTracks()   const;
    Collection<TMBVertex>    getPrimaryVertices() const;
    // [...many more]

    const TMBMet    *getMet() const;
    const TMBGlobal *getGlobal() const;
```

For a simple object that has its own branch (like `TMBGlobal`) we can just ask for a pointer to it. For branches which consist of multiple objects (e.g. all muons), the class returns a `Collection<T>` object. This is a lightweight wrapper around the internal `TClonesArray` implementation that makes it look like an STL container. The usual methods like `begin(), end()` and `size()` are available, as well as the `[]` operator.

```
root [4] tree->GetEntry(0);
root [5] event.getMuons().size();
(const unsigned int)5
root [6] cafe::Collection<TMBMuon> muons = event.getMuons();
root [7] muons.size()
(const unsigned int)5
root [8] const TMBMuon& muon = muons[0]
root [9] muon.Pt()
1.9234234
```

Don't be afraid to copy `Collection` objects around, internally they are not more than a pointer and all the data actually remains in the original `Event` object.

If you have a function like

```
#include "cafe/Event.hpp"
#include <iostream>

bool doSomething(cafe::Event& event)
{
    const TMBGlobal *global = event.getGlobal();
    std::cout << "Event = " << global->evtno()
              << std::endl;
    return true;
}
```

you can call it now like this:

```
root [...] .L do_something.C+
root [...] for(Int_t i = 0; i < tree->GetEntriesFast(); i++) {
           tree->LoadTree(i); do_something(event); event.clear();
           }
```

3.3 `cafe::Event` and User Defined Classes [ADVANCED]

By default, the `Event` class knows only about the objects and branches which are part of the CAF classes. In fact `Event` is only a convenience class for the user. Actually, its base class `EventBase` doesn't know anything about the CAF classes at all.

However, it has methods to access arbitrary branches, e.g. for non-standard objects that are not part of the default CAF format.

3.3.1 Simple Objects

Let's assume that you've added a new branch to your root file which contains variables very specific to your analysis. These definitions will never go into the tree use for central production. We first consider the simply case where all these variables are in one object:

```
class MyVariables : public TObject {
public:
      MyVariables();
      int   myVariableA();
      float myVariableB();
private:
      // ... variables
      int   _A;
      float _B;
public:
      ClassDef(MyVariables, 1);
};
```

The name of branch you've chosen to save it in is "MyAnalysis". We explain later how to add a branch to a CAF file, for now we assume your input file already contains these data.

You can access it after you filled the event like this:

```
tree->LoadTree(0);
const MyVariables *var = event.get<MyVariables>("MyAnalysis");
```

The result will be a NULL pointer if the branch does not exist.

If you don't like the use of templated member functions and specifying the branch name every time, you should probably write a little helper function:

```
const MyVariables *getMyVariables(const cafe::Event& event)
{
   event.get<MyVariables>("MyAnalysis");
}
```

and use that in your code.

3.3.2 `TClonesArrays` of Objects

If you want to store a branch with more than one object, you should preferably put it into a `TClonesArray`. The recipe is very similar to the one above, except that now you call `getCollection()` and the return type is a `Collection<T>` where `T` is your type.

Assuming you have this definition:

```
class MyParticle : public TPhysObject {
```

```
public:
     MyParticle();
     float Pt() const;
private:
     float _pt;
public:
     ClassDef(MyParticle, 1);
};
```

you would access it like this:

```
Collection<MyParticle> my_particles(event.getCollection<MyParticle>("MyAnalysisParticles");

for(Collection<MyParticle>::iterator it = my_particles.begin();
    it != my_particles.end();
    ++it) {
    float pt = (*it).Pt();
}
```

Finally, if you prefer to use the `TClonesArray` directly, you can do this as well:

```
const TClonesArray *tca = event.getClonesArray("MyAnalysisParticles");
```

4 The CAF Environment Framework

Putting all your analysis code into a single function or even a `TSelector` method is not very scalable. Usually you have lots of tasks which you want to run either separately or together, maybe even in a different order.

Sharing and reusing code is very difficult in this way, since all you can do is copy and paste other people's code and modify it to make it work in your environment.

The CAF Environment framework provides the common functionality needed to implement most of the above things in a consistent way. By following a few simple rules, you can extend the framework in a well-defined way and mix and match your code with code that other people have written.

The framework also provides services for configuration and transparent access to files via SAM.

4.1 Concepts [BEGINNER]

The basic unit in the framework is a so-called `Processor`. In its most simple form it has a method that takes a `cafe::Event` object and returns a `bool`.

The user can specify a series of `Processor`s that will be executed for each event. If any of them returns `false`, the processing of the event stops at this point.

The `Processor` objects are orchestrated by another entity called a `Controller`. `Controller`s are themselves a `Processor` and they can be nested, i.e. a `Controller` can contain other `Controller`s.

The most common `Controller` is the `Group`. All it does is execute its children in order.

4.2 Writing your own Processors [BEGINNER]

Most of these concepts will become clearer in the following examples. From now on we assume that you are working in a compiled environment rather than the root interpreter.

You should start by creating a new package for these exercises:

```
% cd work
% ctnewpkg -l my_examples
% ( cd include; ln -s ../my_examples/my_examples )
```

Then, as usual, put your header files into the `my_examples/my_examples` subdirectory and the source files into `my_examples/src`. All your source files should be named in the `COMPONENTS` file. In addition you should add `root` to the LIBDEPS file.

Note that we don't have any `bin` directory. As you will see we won't need it.

4.2.1 Simple Functions

Let's start with a very simple example: we want to print out the event number for each event in a file. This should not be a big programming task. Write a file `src/PrintEventNo.cpp` like this:

```
#include <iostream>
#include "cafe/Event.hpp"
#include "cafe/Function.hpp"

bool PrintEventNo(cafe::Event& event)
{
    const TMBGlobal *global = event.getGlobal();
    std::cout << "Event no: " << global->evtno() << std::endl;
    return true;
}

CAFE_FUNCTION(PrintEventNo)
```

That's it. Notice the `CAFE_FUNCTION` macro at the end. You'll need this to register your function with the cafe environment.

Now type `make` and check that the build system created the `shlib/$SRT_SUBDIR/libmy_example.so` library.

You can now execute this function in the framework by running the following command:

```
% cafe Input: input.root Run: PrintEventNo Packages: my_example
```

Note that there is no need for linking. In fact, we are going to use the same executable `cafe` for all our programs from now on ! It will load our newly compiled shared library at run-time and execute the function we have specified on the command line.

The return value of the function is false if the framework should stop the processing of the event at this point, true otherwise.

The funny arguments are explained later in the section about the configuration file syntax. Basically, the command line consists of a list of (`name`, `value`) pairs. The above says:

- Use `input.root` as the input file.
- Run the `PrintEventNo` function on each event.
- Load the package `my_example` before execution.

4.2.2 Subclasses of `Processor`

The method above is only good enough for very simple tasks. Usually the user needs to have more information about the actual processing, e.g. when new files are opened and closed, etc. In this case, she can inherit directly from the `cafe::Processor` base class and implement any of the virtual methods provided there. One of them is the `processEvent(cafe::Event& event)` method which is called for every event. There are, however, a few more available:

```
class Processor {
   virtual void begin();
   virtual void finish();
   virtual void inputFileOpened(TFile *file);
   virtual void inputFileClosing(TFile *file);
   virtual bool processEvent(cafe::Event& event);
   int  eventCount() const;
};
```

The `begin()` and `finish()` methods are called at the beginning and end of the processing. They can be used to create objects like histograms and save them at the end. The `inputFileOpened()` and `inputFileClosed()` methods inform about the opening and closing of input files.

Finally, the `eventCount` method returns the number of events that this object has processed. This is such a common task, that the framework keeps track of it and uses it at the end to print out the event counts of the various steps. If a `Processor` implements a filter with an analysis cut, this is a quick way to look at the result.

4.3 Input Specifications [BEGINNER]

The `Input:` argument to `cafe` can be any of the following:

- a single filename (optionally prefixed with `file:`), example: `file:data.root` or just `data.root`
- a file with a list of filenames, example: `listfile:mydata.lst`
- a SAM dataset definition, example `sam:My2MUSkim`
- any other prefix known to ROOT, e.g. `http:`, `rootk:` etc.

Note: the files in a file list can in turn have any known ROOT file prefix, but not another file list or a SAM definition.

Some examples:

```
% cafe Input: file:test.root
% cafe Input: sam:MyDataSet
% cafe Input: listfile:myfiles.lst
% cafe Input: rootk://wensley-clued0.fnal.gov/work/wensley-clued0/data.root
```

4.4 Run Specifications [BEGINNER]

The `Run:` argument takes a list of `Processor`s like this:

```
% cafe Input: data.root Run: PrintEventNo,MyFilter,Muon_PT_Histo
```

In the simplest case, the names you specify on the command line are just the names of the functions and classes you wrote for your analysis. In this case it is also the value of the string parameter given to your class in the constructor.

Sometimes, however, it is useful to have more than one instance of the same class run. As long as the classes don't need to be distinguished (e.g for the names of the histograms they write out), you don't care. Otherwise, you need a way to give each a unique name. You do this with the following syntax:

```
% cafe Input: data.root Run: 'Muon_PT_Histo(first),Muon_PT_Histo(second)'
```

This syntax is supposed to mimic calling the constructor of `Muon_PT_Hist` with the string `first` as name. In a C++ program you would say something like:

```
Processor *p = new Muon_PT_Histo("first");
```

If the `Muon_PT_Hist` class uses the string passed to its constructor in naming the histograms, you can distinguish their output, one of them using "first_" as prefix, the other "second_":

```
Muon_PT_Hist::Muon_PT_Hist(const char * name)
{
   string histoName = string(name) + "_muon_pt";
   hist = new TH1F(histName.c_str(), "Muon pT", 0.0, 100.0, 100);
   // ...
}
```

Note that you have to enclose the argument in quotes. This is because the '(' has a special meaning for the shell. If you quote the parameter, you can use either space or comma to separate the `Processor`s.

4.5 Non-Persistent Data in `cafe:Event` [BEGINNER]

A quite common need is the ability to share information between different `Processor`s. Instead of using global variables, the `cafe::Event` class provides a light weight way of passing information from one step to another.

You can store any object (or simple type) in `cafe::Event`, provided that the object has a copy constructor and assignment defined. Each object is stored together with a *key* consisting of a string.

Here is how you store a simple integer number:

```
event.put("myNumber", 5);
```

Any other `Processor` later in the chain can access this value like this:

```
int value;
if(event.get("myNumber", value)) {
  // use 'value'
} else {
  err() << "myNumber not found in Event" << std::endl;
}
```

There is always a copy operation involved in both storing and retrieving the value, so the operation is quite general, but potentially expensive. You might try to circumvent this problem by storing only a pointer in the event. In this case the question of ownership arises. The default behaviour for the `cafe::Event` class is to take ownership of any pointer you pass it. When all `Processor`s have been called for the current event, the `Event::clear()` method is called, which deletes any user pointer stored in the event.

So the following is quite safe as far as memory leaks are concerned:

```
MyObject *obj = new MyObject(...);
event.put("myobject", obj);
```

4.6 Tagging Events [BEGINNER]

A very common piece of information which is passed around between `Processors`s is a *Tag*. This is just a string that marks a specific event. There are default methods to tag an event and check for the existence of tags.

Here is how you tag an event:

```
event.tag("2MU");
```

A tag is just a string which you can attach to an event. You can tag an event with as many tags as you like.

Here is how you check for the existences of a tag:

```
if(event.hasTag("2MU")) {
    // do something with event
}
```

You can also check for the existence of any of a list of tags:

```
std::list<std::string> tags;
// fill tags
if(event.hasTag(tags)) {
   // do something with event
}
```

The latter method works with any STL container. I.e. you can store the tag list in a `list`, a `vector` or a `deque`, the implementation doesn't care.

5 Configuration Files

5.1 Syntax [BEGINNER]

While specifying arguments on the command line is convenient for developement, it can soon become cumbersome if there are more than a few processors. All the command line arguments can be specified in a configuration file which follows the `TEnv` syntax also used in the `rootrc` files.

Here is an example of such a file:

```
cafe.Input:        file:input.root
cafe.Run:          PrintEventNo,Group(test)
cafe.Events:       0

test.Run:          MyFilter,PrintEventNo
```

The relation to the command line arguments should be straightforward. Any entry without a prefix on the command line is interpreted as if it had the `cafe` prefix in the configuration file.

Furthermore any other entry in the configuration file can be overwritten explicitly on the command line with its full name.

The configuration file is searched for in the following places:

- If the first command line argument is a file name, it is used.
- If the `$CAFE_CONFIG` environment variable is defined, the file it points to is used.
- If `./cafe.config` exists, it is used.
- If `$SRT_PRIVATE_CONTEXT/cafe.config` exists, it is used.

5.2 Access from User Code [BEGINNER]

Typically user algorithms also need configuration information that you don't want to hardcode in your source. You can access the configuration file(s) via the `Config` class.

```
{
   //...
   cafe::Config  config("cafe");
   std::string input_spec = config.getString("Input");
}
```

The code above will return the value of `cafe.input`, no matter if it was given in the configuration file or the command line.

You can also store your own information in this file. You should prefix each entry with either your class or function name or the name of your processor instance (i.e. the name you gave it in the run specification).

```
FilterMuonEvents.NumMuons:    2
```

for a class like this:

```
class FilterMuonEvents {
public:
  FilterMuonEvents(const char *name)
  {
     cafe::Config config(name);
     _numMuons = config.getInt("NumMuons");
  }

  bool processEvent(cafe::Event& event)
  {
     return event.getMuons().size() >= _numMuons;
  }

private:
  int _numMuons;
};
```

5.3 Specifying `Groups` [ADVANCED]

There are several special `Processor` classes which provide more structure to the execution of the framework. The most common one is called `Group`. A `Group` object has a `.Run` parameter, just like the top-level `cafe` entry.

What it does is, it executes all its children in sequence. This may not seem much of a deal, but it allows you to structure the execution of the code in such a way, that multiple pieces can be easily added or removed.

The `Processors` that should run as part of a `Group` are specified int the corresponding `Run` parameter:

```
cafe.Run:         Group(Filter),Group(Process),Group(Output)
Filter.Run:       FilterMuonEvents,FilterGoodRuns
Process.Run:      Muon_PT_Histo,Group(PostProcess)
Output.Run:       WriteMyVariables,WriteStandardExtensions
PostProcess.Run: ProduceMyVariables,ProduceStandardExtensions
```

Now, if you don't want to run the `Filter` group, just modify the first line instead of removing multiple entries. `Group`s can be nested arbitrarily.

By default, there is one top-level `Group` called `cafe`.

In addition to the structuring the execution, `Group`s also provide a few more configuration options.

5.3.1 Output Files

If a `Group` has an `.Output` entry in the configuration file, it will open a new root file with that name and make it the current directory. Each `Processor` can ask what its current directory is with the `getDirectory()` method. All the output of a `Processor` should usually go into its current directory. E.g. instead of just creating a histogrom, it should do something like this:

```
getDirectory()->cd(); // change to my directory
hist = new TH1F(...);
```

The default output directory is `gROOT`.

5.3.2 Directories

Each ROOT file can have multiple directories inside it. If the `.Directory` entry of a `Group` is defined, a new subdirectory with the given name is created. Again, this directory is made the default directory for all children.

```
cafe.Run:     Group(One) Group(Two)
cafe.Output:  histos.root

One.Run:       ...
One.Directory: HistosFromOne

Two.Run:       ...
Two.Directory: HistosFromTwo
```

If any of the `Processor`s in `One` and `Two` produce histograms they will end up in the two different folders `HistosFromOne` and `HistosFromTwo` resp. All of the histograms will be part of `histos.root`.

While every `Group` can define its own output file, it is usually most convenient to have just one file, but multiple directories inside.

5.3.3 Debug Output

Each `Processor` has a method `debug()` which returns the current debugging level (a simple integer). The higher the level, the more detailed output should be produced. The debug level can be set via the `.Debug` entry of a `Group` for all its children.

```
MyGroup.Debug: 3
```

and in your code:

```
if(debug() > 2) {
   err() << "Detailed info on what's going on..." << endl;
}
```

5.3.4 More `Processor` Methods

To produce error or standard output, call the `err()` or `out()` methods resp. They will return a `std::ostream&` that you can use as any other.

5.4 Including Configuration Files

You can distribute the information in the configuration file over multiple physical files and include those. Include files can be nested arbitrarily deep. When specifying the file names, you can use environment variables which will be expanded automatically.

```
+cafe.Include:    otherFile.cfg $HOME/myconfig/Config.cfg $MYGROUP/common.cfg
```

Every file is included only once, even when it appears in more than one include directive. Note the use of the plus sign to extend the definition of `cafe.Include`. This allows to have multiple such definitions in different files which will be all concatenated.

6 Extending the CAF Tree [ADVANCED]

Sometimes the information in the CAF tree is not enough for a given analysis, or a group of people wants to share derived variables without recalculating them every time they need them. In these cases the CAF tree should be extended.

Let's assume the following: we want to produce some additional information for every event that we see in our `Processor` and write it to a new output file. For this purpose we create a `TTree` and add a new branch to it.

```
#include "cafe/Processor.hpp"
#include "MyVariables.hpp"

class MyProducer : public Processor {
public:

   void inputFileOpened(TFile *input_file)
   {
       std::string input_name = input_File->GetName();
       input_name = input_name.substr(0, input_name.rfind('.')) + "-myvar.root";
       outfile = new TFile(input_name.c_str(), "RECREATE");
       tree    = new TTree("MyTree");
       tree->Branch("MyAnalysis", "MyVariables", & myVariables, 32000, 99);
```

```
   }

   void inputFileClosing(TFile *input_file)
   {
       outfile->Write();
       outfile->Close();
       delete outfile;
   }

   bool processEvent(cafe::Event& event)
   {
       // calculate myVariables
       tree->Fill();
   }

public:
   TFile       *outfile;
   TTree       *tree;
   MyVariables *myVariables;
};
```

In this example we create for every input file a new file called after the original name and `-myvar.root` appended.

In a later run, we want to use the information we created together with the original file. The typical way to do this is to add the tree in the new file as a friend, and then use the methods described in 3.3 (Extending cafe::Event).

An example is here:

```
   class AddMyVariables : public Processor {
   public:
      void begin(cafe::Event& event)
      {
          event.addBranch<MyVariables>("MyAnalysis");
      }

      // ...
      void inputFileOpened(TFile *file)
      {
         std::string input_name = input_File->GetName();
         input_name = input_name.substr(0, input_name.rfind('.')) + "-myvar.root";
         myfile = TFile::Open(input_name.c_str(), "READ");
         if(myfile.IsOpen()) {
             if(TTree *mytree = (TTree *)myfile.Get("MyTree")) {
                TTree *caf_tree = (TTree *)file->Get("TMBTree");
                caf_tree->AddFriend(mytree);
             }
```

```
        }
    }

    void inputFileClosing(TFile *file)
    {
        myfile->Close();
        delete myfile;
    }

private:
    TFile *myfile;
};
```

However, you should try to re-use some of the existing utilities described in the next section instead of writing this code yourself.

An alternative to the above is to extend the existing tree with new branches and write the output all to one file. This makes mostly sense if a selection is applied at the same time, so the output tree is different from the input tree. The only difference is that you would clone the input tree before adding your own branch.

```
void inputFileOpened(TFile *file) {
   // [...]
   TTree *caf_tree = (TTree *)file.Get("TMBTree");
   TTree *mytree = caf_tree->CloneTree(0);
   mytree->Branch(...);
   // [...]
   save_addressess = true;
}

bool processEvent(cafe::Event& event) {
    // calculate variables
    mytree->Fill();   // write both event and myVariables
}
```

7 Available Processors [BEGINNER]

This section describes various existing Processor classes that can be re-used by an analyzer. Often they do simple things, but having them available and configurable saves a lot of trivial programming tasks.

7.1 Creating Histograms [BEGINNER]

We have shown an example of how to implement a simple Processor that creates a histogram and write it to a file. This is such a common task, that there is a standard Hist1D package available. You configure it with the following parameters:

```
cafe.Run:         Hist1D(MuonHisto)
```

```
MuonHisto.Draw:   Muon.Pt()
MuonHisto.Select: Muon.Pt() > 1.5 && Muon.Pt() < 100.0 && Muon.Eta() < 2.0
MuonHisto.Title:  "Muon pT"
MuonHisto.Bins:   100 0. 100
```

Creates a new histogram with the given (optional) title. The name of the histogram is taken from the name of the `Processor`, i.e. `MuonHisto` in this case.

Into which file and/or directory will the histogram be written ? That depends on the *current* directory for the `Processor`. The easiest way is to specify the output file in the top level configuration:

```
cafe.Output: myhistos.root
```

All histograms will go into this file. However, it is often useful to structure the output file itself. Every `Group` can actually define a directory in the current file and each of its children will write its histogram into that directory:

```
cafe.Run:    Group(mygroup) ...
cafe.Output: myhists.root

mygroup.Run:       Hist1D(test)
mygroup.Directory: MyDirectory

test.Draw:   ...
```

In this case the `test` histogram will be stored in the `MyDirectory` folder inside the `myhistos.root` file.

7.2 Hist2D and Hist3D

There are also 2 and 3-dimensional histograms available. The `.Draw` entry will use two or three specifications, separated by a colon, and the `Bins:` entry takes 6 or 9 parameters, 3 for each dimension (number of bins, minimum, maximum).

```
cafe.Run:   Hist2(test)
test.Draw:  Muon.Eta():Muon.Pt()
```

7.3 Selecting Events [BEGINNER]

```
cafe.Run:            Select(DimuonEvents)
DimuonEvents.Select: Muon[0].Pt() > 10.0 && Muon[1].Pt() > 5.0
```

The expression in the `DimuonEvents.Select` entry follows the syntax used by the `TTree::Draw()` command in ROOT. If it evaluates to false, the `Processor` rejects the event.

7.4 Selecting Events by Trigger [BEGINNER]

```
cafe.Run:              Trigger(DimuonEvents)
DimuonEvents.Triggers: 2MU_A_L2M0 2MU_A_L2ETAPHI
```

Select events by trigger names. While you can use the `Select` Processor as well, this version is much easier to use and faster.

7.5 Adding Friends [BEGINNER]

```
cafe.Run:      Friend(MyFriend)
MyFriend.File: %f-myvars.root
MyFriend.Tree: MyTree
```

Whenever a new input file is opened, this package will try to open a corresponding file whose name is derived from the input file name (the `%f`), plus the `-myvars.root` suffix. It then adds the `TTree` as a friend to the standard tree, so all branches are transparently available.

7.6 Conditional Execution [BEGINNER]

```
cafe.Run:          If(GoodEvent)

GoodEvent.Select:  Muon[0].Pt() > 15.0 && Muon[1].Pt() > 10.
GoodEvent.Then:    ProcessGood,Passed
GoodEvent.Else:    ProcessBad,Passed
```

This controller will evaluate the `GoodEvent.Select` expression, and execute the Processors in the `Then` variable if it true. Otherwise, the Processors in the `Else` variable are executed.

7.7 Permutations [ADVANCED]

```
cafe.Run:             BadRuns BadLBNs Permute(AllPermuations)
AllPermutations.Run:  Select(pTCuts) Select(MetCut) Select(deltaPhiCut)
```

This controller will execute all permutations of its children. It does this by creating a new `Controller` for every permutation. Furthermore, each controller will have a different default directory, so histograms created in the children will end up in different places.

7.8 Creating Subsets of Particles [ADVANCED]

```
cafe.Run:            SelectObjects(GoodJets)

GoodJets.Select:     Jets.Pt() > 10.0 && Jets.emfrac() < 0.15
GoodJets.ToBranch:   "GoodJets"
GoodJets.FromBranch: "Jets"
```

7.9 Writing out Events [BEGINNER]

```
cafe.Run:        Select(GoodEvents),Write(MyFile)

GoodEvents.Select: Muon.nseg() == 3
MyFile.File:       MySkim.root
```

Write out events into the specified file. Note that this is different from the `cafe.Output` file which is used to store histograms. The `Write` object will store the events themselves.

7.10 Controller/Group [ADVANCED]

Apart from executing the methods of its children, a `Controller` provides a number of additional features. The only required configuration option is `Run`, all the others are optional.

```
cafe.Run:  Group(test)

test.Run:       Proc1,Proc2,Proc3
test.Output:    results.root
test.Directory: Results
test.Debug:     2
```

The `Output` parameter allows you to create a new ROOT file. Typically, this is only done on the top-level.

The `Directory` parameter allows to create a new directory in the current file. This can be used in intermediate `Controller`s to differentiate between different execution paths.

The `Debug` parameter sets the debug level for all its children. This level can be checked inside a `Processor` via the `debug()` method.

8 A Quick Tour through Typical Use Cases

The following we give small but complete configuration files to do some common tasks.

8.1 Making a Selection by Triggers

```
cafe.Run:           Trigger(MyTrigger) Passed
MyTrigger.Triggers: 2MU_A_L2M0 2MU_A_L2ETAPHI
```

8.2 Making a Selection by Physics Objects

```
cafe.Run:           Select(MySelection) Passed
MySelection.Select: Muon.isTight() && Muon.nseg() == 3
```

8.3 Creating a Histogram

```
cafe.Run:      Hist1D(MyHisto)
cafe.Output:   histos.root
MyHisto.Draw:  Muon.Pt()
MyHisto.Bins:  0 0. 100.
```

8.4 Writing Events

```
cafe.Run:           Select(MySelection) Write(MyWrite)
MyWrite.File:       MySkim.root
MySelection.Select: Muon.isTight && (abs(Muon.nseg()) == 3)
```

9 Appendix

9.1 Using the `cafe::Event` with `TSelector` [ADVANCED]

An alternative and more elaborated way is to define your own `TSelector` class and use it together with `cafe::Event`.

```
#include "TSelector.h"
#include "TFile.h"
#include "TH1F.h"
#include "cafe/Event.hpp"
#include "tmb_tree/TMBMuon.hpp"


   class MyAnalysis : public TSelector {
   public:
      MyAnalysis()
          : tree(0),
            output_file(0),
            ptcent(0)
      {
      }

      void Init(TTree *new_tree)
      {
          tree = new_tree;
          event.setBranchAddresses(tree);
      }

      void Begin(TTree *tree)
      {
```

```
          Init(tree);

          // Create histograms
          output_file = new TFile("plots.root", "RECREATE");
          ptcent = new TH1F("ptcent", "Central Muon pT", 100, 0.0, 100.0);
      }

     Bool_t ProcessCut(Long64_t entry)
     {
          event.clear();

          tree->GetEntry(entry);
          // Maybe make a selection, return kFALSE
          // if event should not be processed.
          return kTRUE;
     }

     void ProcessFill(Long64_t entry)
     {
          using namespace cafe;

          // Fill histograms
          Collection<TMBMuon> muons = event.getMuons();
          for(Collection<TMBMuon>::iterator it = muons.begin();
              it != muons.end();
              ++it) {
//                ptcent->Fill((*it).Central.Pt());
                ptcent->Fill((*it).Pt());
          }
     }

     void Terminate()
     {
          // Write histograms
          output_file->Write();
          output_file->Close();
          delete output_file;
     }
   private:
       TTree       *tree;
       cafe::Event event;
       TFile       *output_file;
       TH1F        *ptcent;
   };
```

You can either compile these classes e.g. with:

```
root [0] .include $SRT_PRIVATE_CONTEXT/include
root [1] .include $SRT_PUBLIC_CONTEXT/include
root [2] TMBMuon m;
root [3] .L MyAnalysis.C+
```

Step 2 is to trigger the loading of all necessary libraries before you compile the code. Alternatively, you can load the required libraries explicitly:

```
root [2] gSystem->Load("libPhysics.so")
root [3] gSystem->Load("libmet_util.so")
root [4] gSystem->Load("libtmb_tree.so")
```

Then pass a pointer to the object to the `TTree::Process` method:

```
root [0] MyAnalysis *analyse = new MyAnalysis;
root [1] tree->Process(analyse)
```

or you can pass the file name directly to the method:

```
root [0] .include $SRT_PRIVATE_CONTEXT/include
root [1] .include $SRT_PUBLIC_CONTEXT/include
root [2] tree->Process("MyAnalysis.C+");
```

9.2 Controllers [ADVANCED]

There is a special type of `Processor` called a `Controller`. What it does is, it creates a new processing chain below itself and calls each of its children in turn when any of its methods are called.

`Processor`s can be added to a `Controller` in a variety of ways. E.g. you can add a function or object directory to it, if you have access to them:

```
cafe::Controller contr("top");
contr.add(PrintEventNo);
contr.add(new MyProcessor("proc"));
```

Alternatively, you an add a `Processor` *by name*. This assumes that the original function or class is known to ROOT.

```
contr.add("PrintEventNo");
contr.add("Muon_PT_Histos");
contr.add("MyProcessor");
```

Now calling `contr.processEvent(event)` will call the corresponding method on all the registered objects in the order they were added. In practice, you don't need to know any of this, since the framework will use the configuration file to create the proper sequence of `Processor`s for you.

`Controller`s can be nested arbitrarily deep. When you specify a `Controller` on the command line, you should give it a name. You can then tell it which `Processor`s to run by specifying another `Run` parameter.

Instead of spelling out the whole class name, you can also use the short alias `Group`, or simply nothing before the `(`:

```
% cafe Run: 'Group(test)'  test.Run: PrintEventNo,Muon_PT_Histos,MyProcessor
% cafe Run: (test)  test.Run: PrintEventNo,Muon_PT_Histos,MyProcessor
```

Theses short cuts exist because `Controller`s are very common to structure the execution of the analysis.

9.3 The default `RunController` [ADVANCED]

There is one special subclass of `Controller` named `RunController`. It can be used to run over an arbitrary list of input files with a user specified list of `Processor`s. This is basically the `main()` routine of the `cafe` program.

The additional method of interest to the user is `RunController::Run(const std::string& input)`. One can call it with a URL to specify a single file, a list of files or a SAM definition. E.g.

```
int main()
{
    RunController top("main");
    top.add("Muon_PT_Histo");
    top.Run("listfile:files.txt");
}
```

However, there is rarely need to do this yourself, since this is exactly what the `cafe` executable is doing for you.