Table of Contents

A.7 Basic C++ in CMSSW context

Complete:
Detailed Review status

Goals of this page:

This page explains the very basic C++ concepts in the CMSSW context. It goes through a simple analyzer class and explains line by line the C++ elements. After reading this page, you should be able to distinguish the pure C++ features from the adds-on provided by CMSSW framework.

Note that to learn C++, reading this is not enough. Please refer to the abundant material available for this purpose: .

Contents

Introduction

C++ in CMSSW is not different in C++ in general. However, for a new-comer to C++ and to CMSSW it is often difficult to distinguish the features belonging to the C++ programming language and the features added in the CMSSW framework in a given example code. This page is designed to give the basic knowledge of the most common C++ components used in a basic analyzer code example.

Class structure

The CMSSW is an object-oriented framework which consists of **classes**. A class is a data structure which can hold data and functions. An **object** is an instantiation of a class.

We take as an example a simple analyzer class PatBasicAnalyzer which resides in PhysicsTools/PatExamples/plugins/PatBasicAnalyzer.cc. To follow this page you should open in it a new window.

When a CMSSW executable is run with this analyzer module compiled, we can configure the CMSSW framework to take care that our analyzer class is instantiated for each event and that the event information is properly passed to the analyzer.

All information how to write and configure a simple analyzer is provided in WorkBookWriteFrameworkModule, all functional details about this specific analyzer are given WorkBookPATAccessExercise, here we concentrate uniquely on the C++ code elements.

In the following, we will go through the two distinct parts of the example class: the class declaration and the class implementation.

Class declaration

All classes have a class declaration which starts with the keyword `class`. In our example we have

```
class PatBasicAnalyzer : public edm::EDAnalyzer {
 ...
};
```

Most CMSSW classes have the class declaration in a so-called header file with suffix .h in the `interface` directory of each package in the cvs code repository. Our example class has the header part in the same file with the rest of the class implementation. This is often the case for test programs.

Inheritance

After the class name, `: public edm::EDAnalyzer` means that our class **inherits** from another CMSSW class called edm::EDAnalyzer. The derived class inherits the properties of the base class.

- See inheritance in the PatBasicAnalyzer class documentation.
- Find more about inheritance: .

Namespace

`edm::` in this class name is a C++ syntax meaning **namespace** which can be used to group entities under a specific name (here `edm`, a namespace defined in CMSSW referring to event data model), often used to prevent clashes.

- Find more about namespaces: .

Include directives

The class declaration starts with a series of `#include` statements:

```
#include <map>
#include <string>

#include "TH1.h"

#include "FWCore/Framework/interface/Event.h"
#include "FWCore/Framework/interface/EDAnalyzer.h"
#include "FWCore/Utilities/interface/InputTag.h"
#include "FWCore/ParameterSet/interface/ParameterSet.h"
#include "FWCore/ServiceRegistry/interface/Service.h"
#include "CommonTools/UtilAlgos/interface/TFileService.h"
```

These statements tell to the pre-processor to include the file indicated because our class will use its functionality later. These can be standard libraries in C++ as `map` or `string`, ROOT elements as `TH1.h` or CMSSW classes.

Public and private members

The class declaration declares the members of the class which can be public (accessible from anywhere where the object is visible) or private (accessible only from within other members of the same class). The are member functions (with brackets `()` after the function name with or without arguments) and data members of different types.

```
class PatBasicAnalyzer : public edm::EDAnalyzer {
```

```
public:
  explicit PatBasicAnalyzer(const edm::ParameterSet&);
  ~PatBasicAnalyzer();

private:

  virtual void beginJob() ;
  virtual void analyze(const edm::Event&, const edm::EventSetup&);
  virtual void endJob() ;

  // simple map to contain all histograms;
  // histograms are booked in the beginJob()
  // method
  std::map<std::string,TH1F*> histContainer_;

  // input tags
  edm::InputTag photonSrc_;
  edm::InputTag elecSrc_;
  edm::InputTag muonSrc_;
  edm::InputTag tauSrc_;
  edm::InputTag jetSrc_;
  edm::InputTag metSrc_;
};
```

Our example class has only two public member functions, the constructor and the destructor.

Constructor

The **constructor** must have the same name as the class and it is automatically called whenever a new object of this class is created. Our constructor passes the parameters in form of `const edm::ParameterSet&` which will contain the values to be initialized when the new object is created.

`edm::ParameterSet` is a CMSSW class which declaration was included in the include statements of our class. The argument is passed as a **constant reference** (`&` indicating the reference) which means that the function cannot modify the value of the argument.

- See an example on passing the arguments.
- Find more about passing by reference: .

Destructor

The destructor starts with ~ and it is automatically called when an object is destroyed.

Member functions

All member functions are declared in the class declaration. As the PatBasicAnalyzer is a derived class inheriting from the the base class, it inherits all members of the base class. Some functions of the base class are **virtual** and they have empty body in the base class, and they will be given a specific meaning the the derived class.

The keyword `virtual` in

```
virtual void beginJob();
  virtual void analyze(const edm::Event&, const edm::EventSetup&);
  virtual void endJob() ;
```

indicates that the functions `beginJob`, `analyze` and `endJob` are already declared in the base class (from which our analyzer inherits) and they will be implemented in our example class. This keyword would not be necessary in our class anymore but its presence reminds us that that these functions are inherited from the base

class.

- Find more about virtual functions: .

The function `analyze` takes two arguments `edm::Event` and `edm::EventSetup` which pass the information about the event to the analyzer. These arguments are CMSSW framework classes and during the execution of our program the framework will take care of passing all needed information through these classes to our analyzer and we do not have to worry about it.

The keyword `void` indicates that these functions do not return any value.

Data members

Follows a list of data members. The `histContainer_`

```
std::map<std::string,TH1F*> histContainer_;
```

is of type `map`. This a container type available in the standard C++ libraries which was included in the beginning with `#include <map>`. The namespace `std::` indicates that we are explicitly using `map` of the standard library rather than some other `map`. A map is a pair of values and here defined to consist of a string `std::string` - yet another component from the C++ standard libraries - and of a ROOT histogram `TH1F` which was made available by adding `#include "TH1.h"`. When indexed by a value of the first type (called **key** - here the string), a map returns the corresponding value of the second type (here the histogram).

The asterix symbol `*` after the histogram type `TH1F` indicates that the map contains **pointers** to the histograms, i.e it will hold the address of the histogram.

- Find more about pointers: .

It is the programmer's choice to use the map container for histograms in this analyzer. When you write your own analyzer, you can do store your histograms differently. You could also define a single histogram, for example `TH1F* jetTowers_`. Whatever form you choose, they need to be declared in the proper form in the class declaration.

- Find more about C++ standard containers: .

In the following block, several input tags are defined:

```
edm::InputTag photonSrc_;
  edm::InputTag elecSrc_;
  edm::InputTag muonSrc_;
  edm::InputTag tauSrc_;
  edm::InputTag jetSrc_;
  edm::InputTag metSrc_;
```

These will be useful as they will allow us to change to a different collection of particles without recompiling the code. The use of these input tags is explained in WorkBookPATAccessExercise.

Class implementation

After the class declaration, the member functions are implemented. Most CMSSW classes have the class implementation in a file with suffix .cc in the `src` directory of each package.

Before the function implementation, there is another series of `include` directives:

```
#include "DataFormats/PatCandidates/interface/Electron.h"
#include "DataFormats/PatCandidates/interface/Photon.h"
#include "DataFormats/PatCandidates/interface/Muon.h"
#include "DataFormats/PatCandidates/interface/Tau.h"
#include "DataFormats/PatCandidates/interface/Jet.h"
#include "DataFormats/PatCandidates/interface/MET.h"
```

These are needed in the `analyze` function later on.

Each member function of a class is defined with the syntax `classname::functionname { ... }`. In the following, we go through each member function in our example class.

Constructor

The constructor in our example is

```
PatBasicAnalyzer::PatBasicAnalyzer(const edm::ParameterSet& iConfig):
  histContainer_(),
  photonSrc_(iConfig.getUntrackedParameter<edm::InputTag>("photonSrc")),
  elecSrc_(iConfig.getUntrackedParameter<edm::InputTag>("electronSrc")),
  muonSrc_(iConfig.getUntrackedParameter<edm::InputTag>("muonSrc")),
  tauSrc_(iConfig.getUntrackedParameter<edm::InputTag>("tauSrc" )),
  jetSrc_(iConfig.getUntrackedParameter<edm::InputTag>("jetSrc" )),
  metSrc_(iConfig.getUntrackedParameter<edm::InputTag>("metSrc" ))
{
}
```

After the colon, there is a member initialization list (7 lines) and the function itself is empty (between the curly brackets). The data members are initialized to values given to them in the configuration file as explained in WorkBookPATAccessExercise.

- Find more about initialization lists: .

Without going to the details of apparently complicated syntax of each input tag value, we can point out that the input is retrieved by a function called `getUntrackedParameter` belonging to object `iConfig` which is an instance of the class `edm::ParameterSet`.

The function `getUntrackedParameter` is a **template** function, which can act on different types of objects. In this case it acts on the type `edm::InputTag`. The function call is of format `function_name <type> (parameters)`

- Find more about template functions: .

Destructor

The destructor needs to implemented even if it contains no code in our example.

```
PatBasicAnalyzer::~PatBasicAnalyzer()
{
}
```

Booking histograms

We can first have a look at the function `beginJob` which books the histograms which will be then filled in `analyze` function. From our class point of view, it initializes the data member `histContainer_` which was declared in the class declaration.

```
void
PatBasicAnalyzer::beginJob()
{
  // register to the TFileService
  edm::Service<TFileService> fs;

  // book histograms:
  histContainer_["photons"]=fs->make<TH1F>("photons", "photon multiplicity",   10, 0,  10);
  histContainer_["elecs"  ]=fs->make<TH1F>("elecs",   "electron multiplicity", 10, 0,  10);
  histContainer_["muons"  ]=fs->make<TH1F>("muons",   "muon multiplicity",     10, 0,  10);
  histContainer_["taus"   ]=fs->make<TH1F>("taus",    "tau multiplicity",      10, 0,  10);
  histContainer_["jets"   ]=fs->make<TH1F>("jets",    "jet multiplicity",      10, 0,  10);
  histContainer_["met"    ]=fs->make<TH1F>("met",     "missing E_{T}",         20, 0, 100);
}
```

The function uses a service provided by the CMSSW framework. The local variable `fs` is an instance of the class `edm::Service`. `Service` class is again a template class and here is of type `TFileService`. The use of this service is described in SWGuideTFileService.

The data member `histContainer_` was declared as a map of a string and a histogram. A string in C++ is passed within quotation marks as `"photons"`. The syntax to access the elements of different C++ standard library containers (vectors, lists, maps...) is designed to be similar: `container[index]`.

The histograms are booked through the `make` function of `TFileService` as explained in the TFileService documentation.

Analyze data

The CMSSW framework calls the `analyze` function for each event and it is the function where you will add your own analysis code.

First it gets the collections of different reconstructed objects:

```
// get jet collection
  edm::Handle<edm::View<pat::Jet> > jets;
  iEvent.getByLabel(jetSrc_,jets);
```

The access to the jet collection is available because the corresponding class definition in `DataFormats/PatCandidates/interface/Jet.h` was included earlier.

The jet collection is called `jets`. This is a local variable declared of type `edm::Handle` which is a framework class always used to get access to the collection (see a basic example in WorkBookWriteFrameworkModule#GeT) . `edm::Handle` is yet another templated class and the type of the collection is defined between `< ... >`. The syntax of type of the collection is particularly complicated in this example as inside it has another templated class edm::View, a container defined in the CMSSW framework.

The collection is then connected to the data of this particular event in `iEvent.getByLabel(jetSrc_,jets)`. `iEvent` is the reference to the event information which is passed as an argument of the `analyze` function. `jetSrc_`, the 1st argument of `getByLabel` function, is one of the data members of our analyzer class. It was declared as a member of the class so that it can be changed in the configuration file without recompilation of the code. Had it not been a member of the class, it could have been given here as a local variable of type `edm::InputTag` as in WorkBookWriteFrameworkModule#GeT.

In the analyzer code, several control structures (for loops, if statements etc) are used. In the following, we go through these structures in detail.

- Find more about C++ control structures in general: .

For loop

The loop over the reconstructed jets is

```
// loop over jets
  size_t nJets=0;
  for(edm::View<pat::Jet>::const_iterator jet=jets->begin(); jet!=jets->end(); ++jet){
    if(jet->pt()>50){
      ++nJets;
    }
  }
  histContainer_["jets"]->Fill(nJets);
```

The `for` loop structure in C++ is of form `for (initialization; condition; increase) statement;` where a simple statement does not need to enclosed in braces `{...}` (but it can).

A simple loop could be

```
for (int n=0; n<10; ++n) {
    cout << n << ", ";
}
```

with `int n=1` as the **initialization**, `n<10` as the **condition** and `++n` as **increase**. In C++, increase (`++`) and decrease (`--`) operators increase or reduce by one the value stored in a variable. They can be as prefix and as a suffix.

- Find more about C++ operators: .

In our example the **initialization** is `edm::View<pat::Jet>::const_iterator jet=jets->begin()`. All C++ standard library containers share a similar interface with similar access methods. The framework container `edm::View` is designed in same way and we can define an iterator `jet` which gets it initial value by calling the `begin()` function of the jet collection.

- Find more about C++ iterators: .

The **condition** of the statement is `jet!=jets->end()`. Similarly as `begin()`, `end()` is an access function which gives the end of the container. The operator `!=` means "not equal" in C++ and the for loop goes from the beginning to the end of the jet container

Note that as the iterator `jet` is a pointer, we need to access its members with the **arrow** (->) and not with the **dot**.

- Find more about dots and arrows: .

If statement

Inside the `for` loop, an `if` statement

```
if(jet->pt()>50){
      ++nJets;
    }
```

checks the value of the the jet pt and if it is above the threshold value the variable `nJets` is incremented. Note that `nJets` was declared as size_t which is a C++ standard library type indicating size of objects. `pt()` is an inherited member of the class pat::Jet. You will not find it in the list of the member functions of the `pat::Jet` class itself, but if you have a look at the base class interface reco::Candidate you will find this function there together with many common functions which are applicable to any kind of reconstructed object.

Accessing a class member

After the `do` loop, the histogram `"jets"` is filled with the number of jets above the pt threshold in the if statement. The other histograms are filled with the number of corresponding objects in each event.

```
histContainer_["jets"]->Fill(nJets);

  // do something similar for the other candidates
  histContainer_["photons"]->Fill(photons->size() );
  ...
```

Again, we need to use the arrow to get the size of the collection.

To be noted, when filling the missing energy, a check is made whether the met container is empty:

```
histContainer_["met"  ]->Fill(mets->empty() ? 0 : (*mets)[0].et());
```

This is done using the conditional operator `?`. If we have a conditional expression `expression1 ? expression2 : expression3`, expression1 is evaluated first. If it is true, expression2 is the value the whole expression, if it is false, it is expression3.

In the expression `( *mets)[0].et())`, the operator `*` is a **dereferencing** operator and `*mets` gives the value of the variable to which `mets` points to. We can therefore get the first element of this container (which in the case of missing et is the only element of this container) and get its transverse energy.

- Find more about dereferencing: .

Declaring plugins to CMSSW

Note that the last two lines in the file

```
#include "FWCore/Framework/interface/MakerMacros.h"
DEFINE_FWK_MODULE(PatBasicAnalyzer);
```

are necessary to declare the analyzer "plugin" to the CMSSW framework. A macro `DEFINE_FWK_MODULE` defined in CMSSW is used for it.

- Find more about declaring plugins: SWGuideDeclarePlugins

Review status

Reviewer/Editor and Date (copy from screen)	Comments
KatiLassilaPerini - 16 Mar 2010	created the pages

Responsible: KatiLassilaPerini
Last reviewed by:

This topic: CMSPublic > WorkBookBasicCPlusPlus
Topic revision: r27 - 2014-07-14 - LucasBrito