Writing an IceTray Module
Example: Linefit Reconstruction

Tyce DeYoung
University of Maryland

IceCube

IceCube Collaboration Meeting
Uppsala, Sweden
October 8, 2004

What Do We Need to Do?

- IceTray Module
  - What types of data is it interested in? Physics
  - Will it need access to any special services? No
  - Is there anything it should do besides process data?

    Yes: Configuration
- Output Records
  - What type of data will I write to the event?

    Tracks and RecoResults
  - What should I use as a base for my data type?

    BasicMuon and RecoResultSingleTrack

Output Records

- The Linefit reconstructs all events as being produced by a single track
  - RecoResultSingleTrack represents exactly this – we can use it "off-the-shelf"
- The track result it produces is a plain old muon track (infinite, no energy information) with one new parameter
  - "Velocity" vector: related to amount of scattering, not interpreted as a "real" velocity
  - Write a new class, inheriting from I3BasicMuon

Writing a New Data Class

- In-file documentation (doxygen/CVS)
- Inherit from the base class (I3LineFitTrack)
- Constructor and destructor (C++)
- Add our new variables
  - Variables are private
  - Need to add accessor methods
- For particle types, need to add CopyTo method to copy the new variables
- ToStream, for printing the variable
- Root macro and Ptr typedef

Linefit Track Class

```
/**
   Copyright (C) 2004
   The IceCube Collaboration
   $Id: $

   @file I3LineFitTrack.h
   @version $Revision: $
   @date $Date: $
   @author deyoung
*/

#ifndef I3LINEFITTRACK_H_INCLUDED
#define I3LINEFITTRACK_H_INCLUDED


#endif // I3LINEFITTRACK_H_INCLUDED
```

Doxygen File Header

CVS Insertion Points

Include Block (Code goes here)

Inheriting from BasicTrack

```
#ifndef I3LINEFITTRACK_H_INCLUDED
#define I3LINEFITTRACK_H_INCLUDED

#include "dataclasses/I3BasicTrack.h"


/**
* @brief A track class to store the
* results of the linefit
*/
class I3LineFitTrack : public I3BasicTrack {


  ClassDef(I3LineFitTrack, 1);
};

#endif // I3LINEFITTRACK_H_INCLUDED
```

Inherit from the BasicTrack class from dataclasses

Doxygen Class Description

Root macro with version number

Adding New Variables

```
class I3LineFitTrack : public I3BasicTrack {

private:

  Double_t fLFVel;
  Double_t fLFVelX;
  Double_t fLFVelY;
  Double_t fLFVelZ;

  ClassDef(I3LineFitTrack, 1);
};
```

Variables are *private!*
No direct access

New variables

Root types
(Please use!)

Accessing New Variables

```
private:
   Double_t fLFVel;
   Double_t fLFVelX;
   Double_t fLFVelY;
   Double_t fLFVelZ;

public:
   virtual Double_t GetLFVel() const {return fL
   virtual void SetLFVel(Double_t arg) {fLFVel

   virtual Double_t GetLFVelX() const {return fLFVelX;}
   virtual void SetLFVelX(Double_t arg) {fLFVelX = arg;}

   virtual Double_t GetLFVelY() const {return fLFVelY;}
   virtual void SetLFVelY(Double_t arg) {fLFVelY = arg;}

   virtual Double_t GetLFVelZ() const {return fLFVelZ;}
   virtual void SetLFVelZ(Double_t arg) {fLFVelZ = arg;}
```

`virtual`: can be reused easily

Access Methods

Use `const` where appropriate

Constructor and Destructor

```
private:

  I3LineFit(const I3LineFit& source);
  I3LineFit& operator=(const I3LineFit& source);

public:

  I3LineFitTrack() :
      fLFVel(NAN);
      fLFVelX(NAN);
      fLFVelY(NAN);
      fLFVelZ(NAN);
  {};

  virtual ~I3LineFitTrack() {};
```

Assignment op. & copy constructor

Constructor: initialize variables

Use NAN, no magic numbers

Destructor (virtual!)

Copying Particles

- Particle classes are hierarchical
  - Base classes have very little information
  - Lower levels have additional parameters
- Want to be able to copy "everything that's relevant" to a new object
  - Using a LineFitTrack as the base for a LikelihoodTrack, for example
  - Copy direction, position, but not "velocity"
- Not in C++ – need to implement manually
  - Recursive: copy base class, then my class' special parameters if copying to another object of my class

Copying Tracks

```
virtual Double_t GetLFVelZ() const { return fLFVelZ; }
virtual void SetLFVelZ(Double_t arg) { fLFVelZ = arg; }
```

Must be `virtual`!

`const`!

```
virtual void CopyTo(I3Particle& destination) const {

  I3BasicTrack::CopyTo(destination);

  I3LineFitTrackPtr linefit =
      dynamic_pointer_cast<I3LineFitTrack>
                          (&destination);

  if (linefit) {
      linefit->fLFVel = fLFVel;
      linefit->fLFVelX = fLFVelX;
      linefit->fLFVelY = fLFVelY;
      linefit->fLFVelZ = fLFVelZ;
  }
}
```

Copy parameters from base class

Check whether target is another LineFitTrack

Printing with ToStream()

```
    if (linefit) {
       linefit->fLFVel = fLFVel;
       linefit->fLFVelX = fLFVelX;
       linefit->fLFVelY = fLFVelY;
       linefit->fLFVelZ = fLFVelZ;
    }
  }

  virtual void ToStream(ostream& o) const
    I3BasicTrack::ToStream(o);
    o << "LFVel: " << fLFVel
      << " LFVelX: " << fLFVelX
      << " LFVelY: " << fLFVelY
      << " LFVelZ: " << fLFVelZ
      << "\n";
  }

  ClassDef(I3LineFitTrack, 1);
};
```

Must be `virtual`!

Works on any output stream

Use base class to print all inherited info

Enable Smart Pointers

```
  virtual void ToStream(ostream& o) const {
     I3BasicTrack::ToStream(o);
     o << "LFVel: " << fLFVel
       << " LFVelX: " << fLFVelX
       << " LFVelY: " << fLFVelY
       << " LFVelZ: " << fLFVelZ
       << "\n";
  }

  ClassDef(I3LineFitTrack, 1);
};

typedef PtrPolicy<I3LineFitTrack>::ThePolicy
  I3LineFitTrackPtr;

#endif // I3LINEFITTRACK_H_INCLUDED
```

Define the "Ptr" for the new class

Writing the IceTray Module

- Now we have the classes we'll use to write our results into the event
  - I3RecoResultSingleTrack off-the-rack
  - Our new I3LineFitTrack
- We're ready to write our module to plug into IceTray applications
  - Get most of our functionality from I3PhysicsModule
  - Need to write new code where the default functionality isn't suitable

What Transitions?

Header and Implementation

- In the track class, everything was pretty trivial
  - Basically just a data container
  - We just put everything into one file
- The LineFit module actually *does* something
  - Don't want everything in one file – have to read through the code to find what functions are legit
  - Put the interface in a header file (I3LineFit.h)
  - Implementation is a second file (I3LineFit.cxx)

File Header

```
/**
* Copyright (C) 2004
* the IceCube Collaboration
* $Id: $
*
* @file I3LineFit.h
* @version $Revision: $
* @date $Date: $
* @author deyoung
*/

#ifndef I3LINEFIT_H_INCLUDED
#define I3LINEFIT_H_INCLUDED


#endif // I3LINEFIT_H_INCLUDED
```

Inheriting the Basic Module

```
#ifndef I3LINEFIT_H_INCLUDED
#define I3LINEFIT_H_INCLUDED

#include "phys-services/I3PhysicsModule.h"

/**
* @brief IceTray module to implement the linefit first
  guess
* routine
*/
class I3LineFit : public I3PhysicsModule {



};

#endif // I3LINEFIT_H_INCLUDED
```

`I3PhysicsModule` is like `I3Module` but includes convenience functions

Constructors & Transitions

```
class I3LineFit : public I3PhysicsModule {
public:
   I3LineFit(I3Context& ctx);
   ~I3LineFit();
   void Configure();
   void Physics(I3Frame& frame);
private:
   I3LineFit();
};
#endif // I3LINEFIT_H_INCLUDED
```

Constructor takes a "context" object

The two transitions we need to implement

Process transitions take a Frame as input

Default constructor to prevent output

Configurable Parameters

```
  void Configure();

  void Physics(I3Frame& frame);

private:

  I3LineFit();

  string fFitName;
  string fInputSelection;
  string fInputReadout;
  Int_t fMinHits;

  ClassDef(I3LineFit,0);

};

#endif // I3LINEFIT_H_INCLUDED
```

Internal variables

Root macro with version number (0 ➔ don't write)

Implementation File

```
/**
* Copyright (C) 2004
* The IceCube Collaboration
* $Id: $
*
* @file I3LineFit.cxx
* @version $Revision: $
* @date $Date: $
* @author deyoung
*/

#include "linefit/I3LineFit.h"
#include "linefit/I3LineFitTrack.h"

using namespace std;

ClassImp(I3LineFit);
```

Include our track and module header files

Access the standard libraries, etc.

Root macro

Other Include Files

```
#include "linefit/I3LineFit.h"
#include "linefit/I3LineFitTrack.h"

#include "icetray/I3TrayHeaders.h"
#include "dataclasses/I3Event.h"
#include "dataclasses/I3Geometry.h"
#include "dataclasses/I3OMResponseMap.h"
#include "dataclasses/I3OMResponse.h"
#include "dataclasses/I3AMANDAAnalogReadout.h"
#include "dataclasses/I3Track.h"
#include "dataclasses/I3Position.h"
#include "dataclasses/OMKey.h"
#include "dataclasses/I3RecoResultSingleTrack.h"
#include "selector/I3OMSlr.h"

#include <iostream>

using namespace std;
```

Main IceTray header file

Dataclass headers

Constructor

```
I3LineFit::I3LineFit(I3Context& ctx) :
  I3PhysicsModule(ctx) {

  AddOutBox("OutBox");

  AddParameter("Name",
     "Name to give the fit the module adds to the
  event",
     "LineFit");

  AddParameter("MinHits",
     "Minimum number of hits: ev th fewe
  will not\n"
     "be reconstructed.", 2);

  fInputSelection.clear();
  AddParameter("InputSelection",
     "OMResponse selector to use for input",
  fInputSelection);

  AddParameter("InputReadout", "Data Readout to use for
  input",
```

Add Outbox: default is 1 Inbox, no Outbox

Parameter name

Default

Description

Configuration

```
I3LineFit::~I3LineFit(){
}
```

Default destructor (Nothing special to clean up.)

```
void I3LineFit::Configure(){

  GetParameter("Name",fModuleName);
  GetParameter("MinHits",fMinHits);
  GetParameter("InputSelection",fInputSelection);
  GetParameter("InputReadout",fInputReadout);
}
```

Parameter name and target location

Physics – The Main Event

```
void I3LineFit::Physics(I3Frame& fra
```

Log message with debug (lowest) priority

```
  log_debug("Entering LineFit Physics().");


  I3Event& event = GetEvent(frame,"Physics");
  I3Geometry &geometry = GetGeometry(frame,
  "Geometry");
```

Get the Physics (event) and Geometry info out of the Frame

Here I would declare the variables I'll use

Getting an OM Selection

```
I3OMSlrPtr slrptr;

if (fInputSelection.empty()) {

   I3OMSlrPtr basic_slrptr(new I3OMSlr);
   slrptr = basic_slrptr;

} else {

   I3Bag& theBag = event.GetBag();
   TObjectPtr objptr = theBag.Get(fInputSelection);
   if (!objptr) { log_error("Specified Selector does not
   exist."); }

   slrptr = roost::dynamic_pointer_cast<I3OMSlr>(objptr);
   if (!slrptr) { log_error("Specified object is not an
   OMSlr."); }
}

I3OMSlr& slr = *slrptr;
```

I'll create a plain OMSlr (basically just an iterator)

Get the selector from the Bag and check that it exists

Cast the TObject I got from the Bag to an OMSlr

Looping over the OMs

```
I3OMResponseMap &om_map = event.GetOMResponseMap();

slr.SetLimit(om_map.end());

for (slr = om_map.begin(); slr != om_map.end(); ++slr)
  {

  log_debug("Looking at OM %i on string %i.", slr-
  >first.GetOM(),
            slr->first.GetStri
  I3OMResponsePtr omr_p = slr-
```

Need to tell the selector about the end of the Map so it doesn't run past the end

`first` and second get OMKey and OMResponse

Getting the Leading Edges

```
I3DataReadoutDict &dict = omr_p->GetDataReadoutDict();
I3DataReadoutPtr readout_ptr = dict[fInputReadout];
```

Get the collection of DataReadouts

I get the DataReadout I want by asking for it by name

```
I3AMANDAAnalogReadoutPtr aarp =
   roost::dynamic_pointer_cast<I3AMANDAAnalogReadout>
   (readout_ptr);
```

Cast the Readout I got to an AMANDA readout (Should check that it succeeded)

```
Double_t time = aarp->GetFirstLE();
```

Getting the OM Position

I use `first` to get the OMKey

```
OMKey om = slr->first;

const I3Position pos = geometry.GetInIceGeometry()[om]-
  >GetPos();
```

I use the OMKey to look up the OM's geometry record

```
Double_t x = pos.GetX();
Double_t y = pos.GetY();
Double_t z = pos.GetZ();
```

Create Result Objects

```
I3RecoResultSingleTrackPtr result (new
   I3RecoResultSingleTrack);
I3LineFitTrackPtr track(new I3LineFitTrack);
```

Create the objects I'll use to store the fit results

```
I3Position ptrack;
ptrack.SetX(Xpos * I3Units::meter);
ptrack.SetY(YPos * I3Units::meter);
ptrack.SetZ(ZPos * I3Units::meter);
track->SetPos(ptrack);

track->SetT(0.);

track->SetZenith(acos( (-1) * velocity[2] / speed)
                 * I3Units::radian);
```

Units!!

Set the track values

Write The Results

```
    log_info("Fit Complete. %s",
        track->ToString().c_str());
```

Use `ToString()` to dump the whole object to the logging stream – calls `ToStream()`

```
    result->SetTrack(track);
    event.GetRecoResultDict()[fModuleName] = result;
```

Add the result to the event

```
    PushFrame(frame,"OutBox");
    log_debug("Exiting LineFit Physics.");
}
```

Put the frame in the Outbox

Documentation

- I've skipped (almost) all of the documentation – <u>not</u> because it's not important!
- Method documentation:
- /**
- * @brief Gives a Foo associated with the given Bar
- * @return The Foo which is of utmost importance
- * @param bar The Bar to get the Foo for. If it's
- * a juice Bar, zero is returned. There's no place
- * for a Bar like that in IceCube software.
- * @todo Find out what happens when the rabbi,
- * priest, and minister enter the Bar.
- */
- FooPtr GimmeSomeLuvin(const Bar& bar) const;

Testing

- Should also write a test suite for your module
- Working now on putting together a package for testing modules