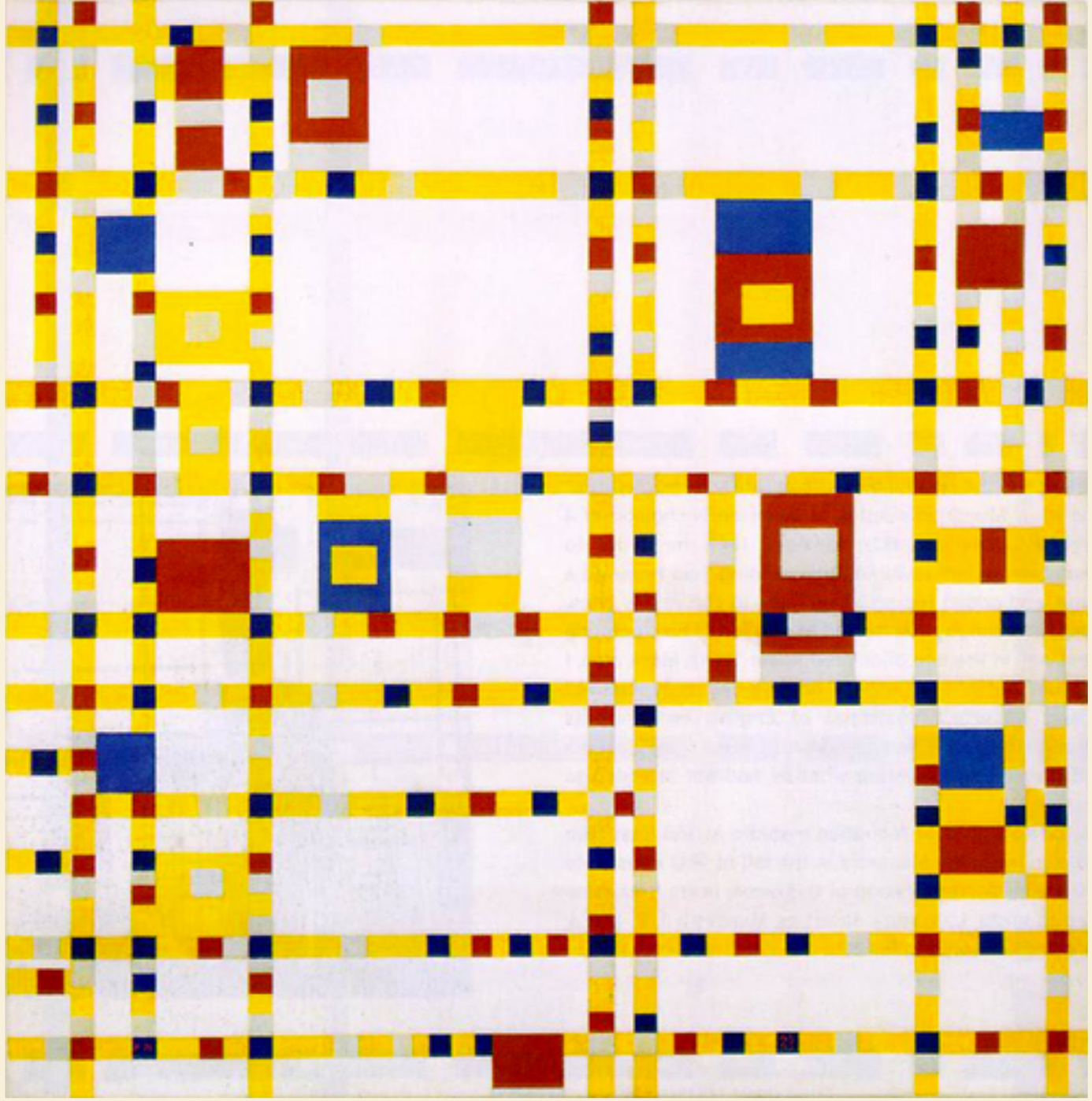


Mondrian, *Broadway Boogie-Woogie*, Museum of Modern Art, NY

Session 10: Data

[art workbook section 16]

Adam Lyon

art/LArSoft School 2015

It's all about the DATA

The framework is all about processing DATA

- **Data I/O – moving data in and out of the framework from/to external storage**
- **Data Creation – Creating data from scratch or derived from existing data**
- **Data Processing – Performing calculations/algorithms with the data**
- **Data Analysis – Summarizing, histogramming**

The framework gives you…

- API: Application Program Interface
- Allows you to find and retrieve data from stores (event, run, subrun) in a standard way
- Serializes the data for I/O [next slides]
- Standardizes access (e.g. API) "Learn once, use everywhere"
- Applies some restrictions on what you can do (uh ok?)
- Mandates some restrictions on the data structure
  (Oh no! I can't try my strange but perhaps awesomely clever idea!
  Breathe deeply - you'll get over it)

You can store fruit by cerealization

Transform the data into a form suitable for storage

You store data by serialization

Transforming in memory data to on storage data
[examples, complications/endianness]

A challenge for C++

- **Store C++ objects such that they can be retrieved and reconstituted as objects later** **[What's wrong with the previous picture?]**
- **If your objects were only primitive types, it would be easy; but we want the flexibility that C++ provides with user-defined types (classes)**
- **We use *Root* for serialization (but we have the capability to use something else someday** **[Why?])**
- **A restriction: No pointers nor references!** **[Why?]**

We use “data products”

“A unit of information that user code may add or retrieve from an event” [art workbook 3.6.4]

C++ objects enclosing related information

Examples:

- **Raw data for a detector subsystem**
- **Hits in a calorimeter *(cal#,x,y,z,E)***
- **Tracks *(x,y,z,px,py,pz)***
- **Electrons, Ws, …**

What does a "data product" contain?

<u>Values</u> (ints, doubles, strings, vectors, other objects) - as private member data

- Hit positions, track parameters, cluster neighbors, Dark Matter mass, g-2 oscillation frequency

<u>Accessor and setter methods</u>

- Lets you look at contents of the object
- Think twice about having setter methods. Values not mandatory by the constructor and deferred to setter methods may be forgotten [take oath→]

<u>Algorithm methods that act on the values in the object</u>

- Think twice before embedding algorithms in the data product. Only put simple, universal algorithms (e.g. invariant mass) in the product. Other algorithms (e.g. track fitting) should be called by art modules [Why?]
- See Data Product Design Guide (more on this in a later talk)

Repeat after me

I will employ features of C++ to help users (and me) avoid mistakes

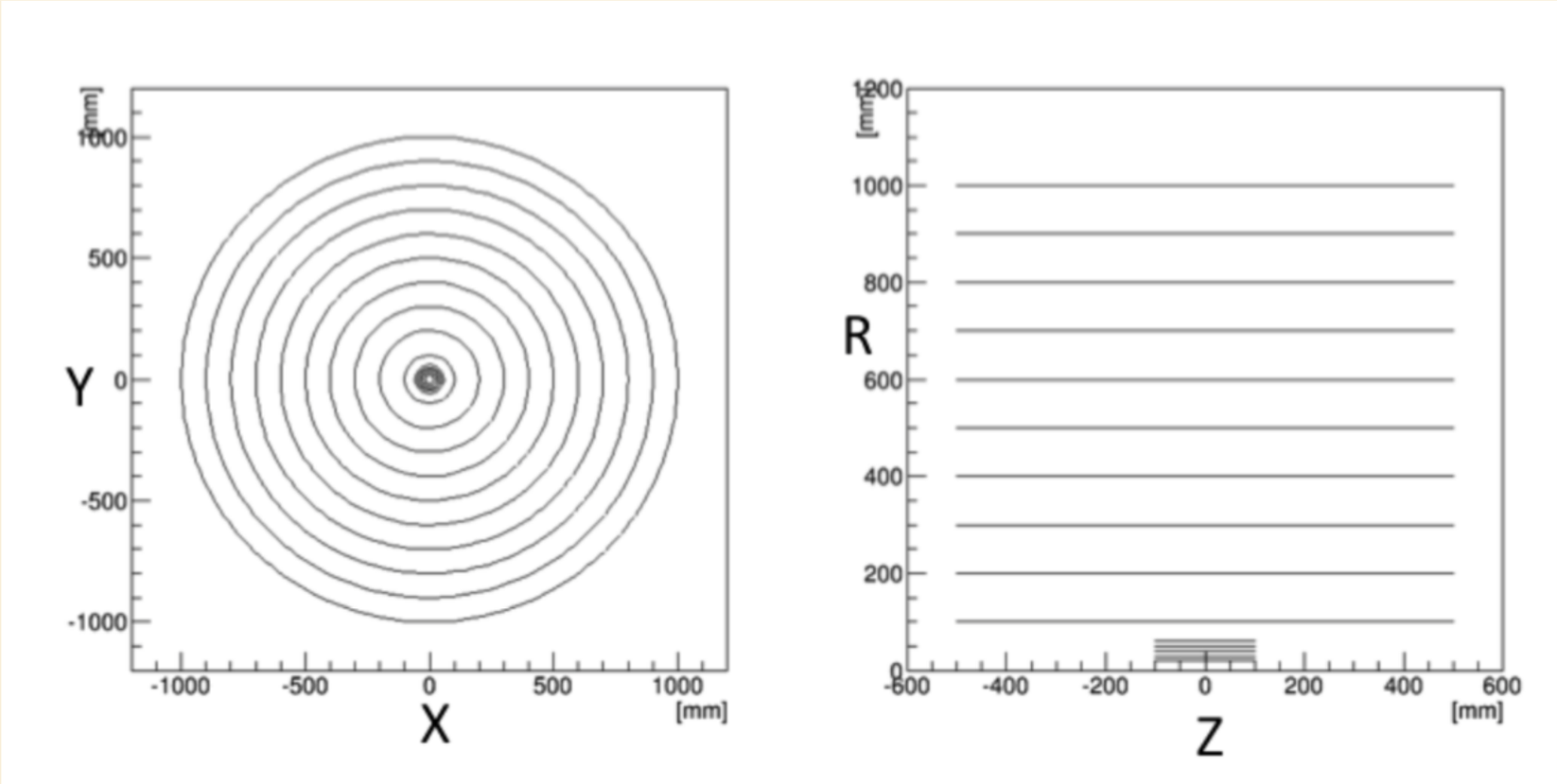
A little extra typing is worth the trouble!

Every mistake you can render nonexistent saves time, money, frustration, …

Here's a story with data products

You are simulating a "Totally Optimal Yak (TOY) Detector"

Consider a cylindrical yak

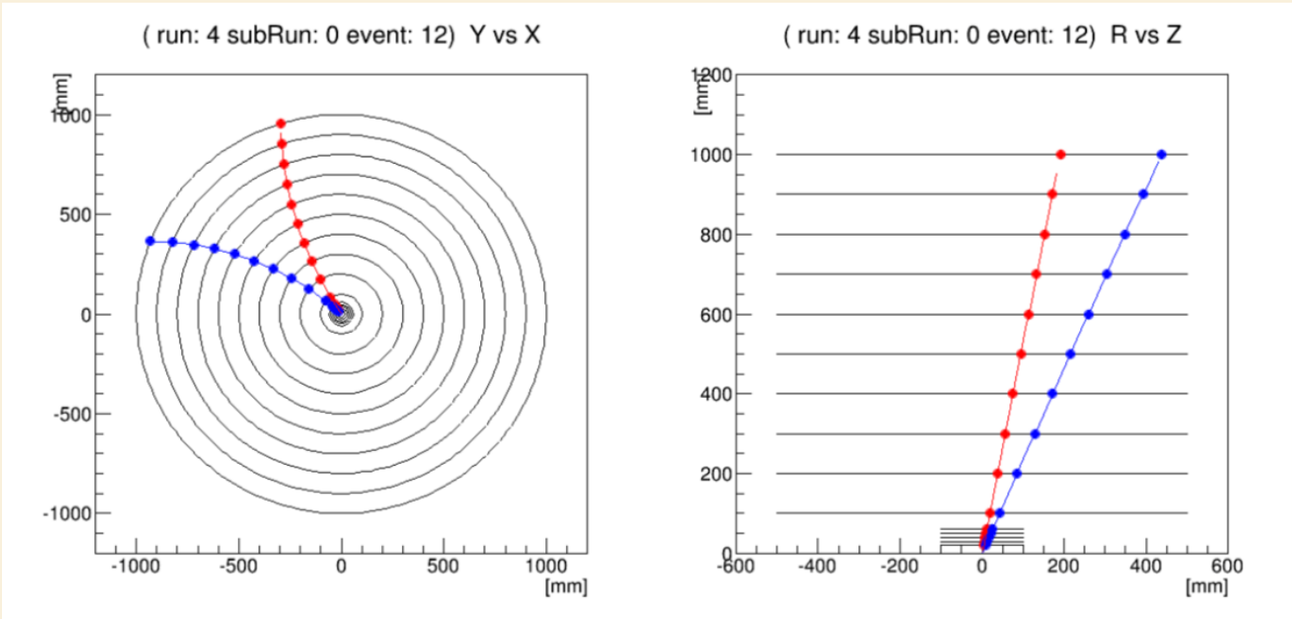


15 concentric massless shells (5 inner closely spaced) in a 1.5 T uniform magnetic field along +z

Simple so you can concentrate on software, not physics

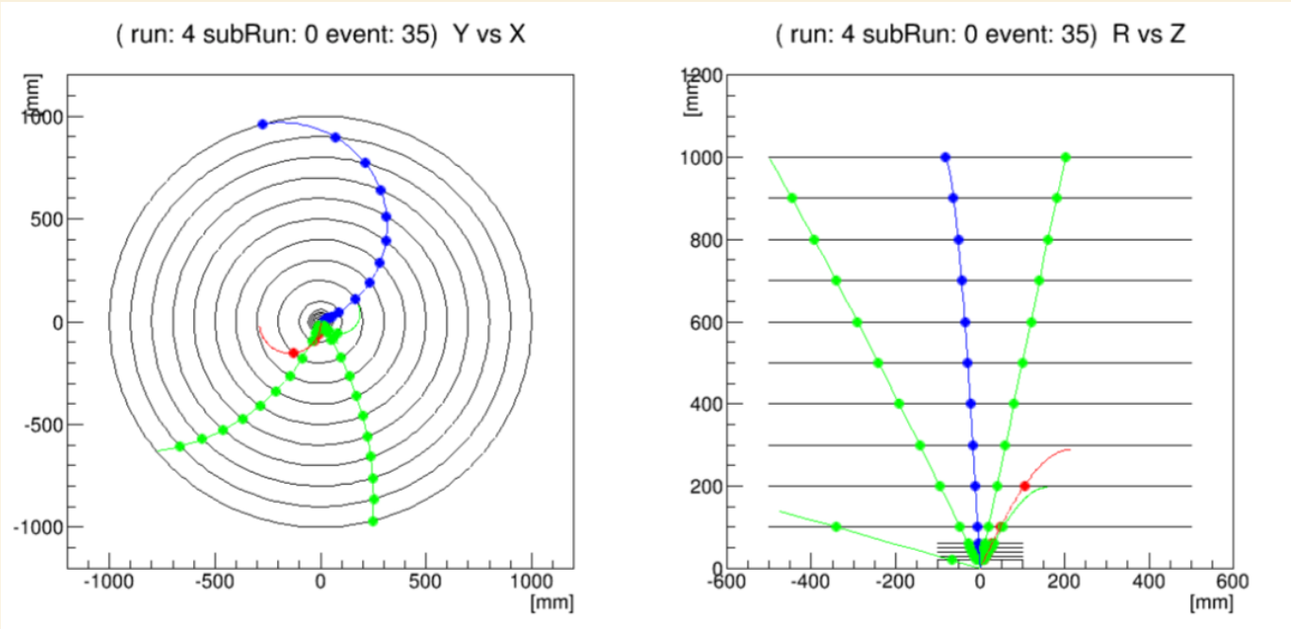
[Your detector will be much more complicated, less totally optimal, and with less yak]

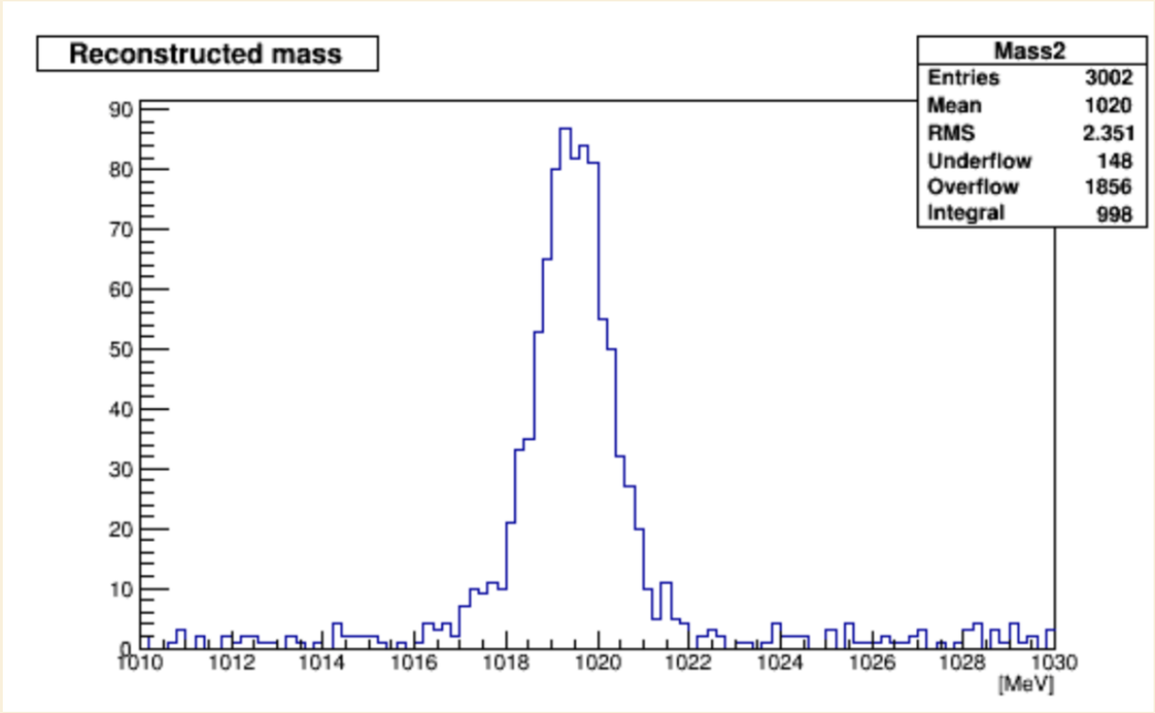
You are generating events



$$\phi \rightarrow K^+K^-$$

with background pion pairs





Your simulation makes data products

Generated particles	`tex::GenParticle`
Intersections (truth hits)	`tex::Intersection`
Reconstructed Hits	`tex::TrkHit`
Fitted tracks	`tex::FittedHelixData`
Detector status	`tex::DetectorStatus`
Summary	`tex::MCRunSummary`

Where are data products?

Can start by looking at the module

See `art-workbook/ReadGenParticles/ReadGens1_module.cc`

Data product should be in a `#include`

```
//  Exercise 6: ReadGenParticles/ReadGens1_module.cc

#include "toyExperiment/MCDataProducts/GenParticleCollection.h"

#include "art/Framework/Core/EDAnalyzer.h"
#include "art/Framework/Core/ModuleMacros.h"
#include "art/Framework/Principal/Event.h"

#include <iostream>
#include <string>

namespace tex {

  class ReadGens1 : public art::EDAnalyzer {
```

Let's look at the data product

```
// $TOYEXPERIMENT_INC/MCDataProducts/GenParticleCollection.h
#ifndef MCDataProducts_GenParticleCollection_h
#define MCDataProducts_GenParticleCollection_h

#include "toyExperiment/MCDataProducts/GenParticle.h"
#include <vector>
namespace tex {
  typedef std::vector<GenParticle> GenParticleCollection;
}

#endif /* MCDataProducts_GenParticleCollection_h */
```

What's in the event isn't one generated particle, but many!

This code defines a typedef for the vector of generated particles (typedef for the vector is a very typical thing to do)

Let's look at the `GenParticle` itself (but remember the data comes out of the event as a vector)

A generated particle

Leaving out some distracting bits…

```
// $TOYEXPERIMENT_INC/MCDataProducts/GenParticle.h

#include "toyExperiment/DataProducts/PDGCode.h"
#include "CLHEP/Vector/LorentzVector.h"
#include "CLHEP/Vector/ThreeVector.h"

namespace tex {
 class GenParticle {
  public:
   GenParticle();

  GenParticle( PDGCode::type                  pdgId,
               art::Ptr<GenParticle> const&   parent,
               CLHEP::Hep3Vector const&       position,
               CLHEP::HepLorentzVector const& momentum );
   PDGCode::type                  pdgId()           const { return  _pdgId;              }
   bool                           hasParent()       const { return  _parent.isNonnull(); }
   bool                           hasChildren()     const { return !_children.empty();   }
   art::Ptr<GenParticle> const&   parent()          const { return  _parent;             }
   children_type const&           children()        const { return  _children;           }
   art::Ptr<GenParticle> const&   child( size_t i ) const { return  _children.at(i);     }
   CLHEP::Hep3Vector const&       position()        const { return  _position;           }
   CLHEP::HepLorentzVector const& momentum()        const { return  _momentum;           }
```

Things to note

Accessors (no setters)

Default constructor and a regular constructor

Only look at the public methods (you can't access the private data/methods - for good reason)

Some people use `struct` instead of `class` [difference?]

`struct` is not recommended

g-2 Calorimeter Truth Hit

```
// gm2ringsim/calo/CaloArtRecord.hh

#include "CLHEP/Vector/ThreeVector.h"

namespace gm2ringsim {
    struct CaloArtRecord {
        int turn;
        int caloNum;
        int trackID;
        float x;        float y;        float z;
        float time;
        float px;        float py;        float pz;
        float energy;
        std::string particle_name;
        int parent_ID;

        CaloArtRecord(){};   // BAD BAD BAD BAD BAD BAD BAD BAD BAD BAD

        CaloArtRecord(int n, int cn, int id,
                      float x, float y, float z, float time,
                      float px, float py, float pz, float e,
                      std::string particleName, int parentID ) :
        turn(n), caloNum(cn), trackID(id),
        x(x), y(y), z(z), time(time),
        px(px), py(py), pz(pz), energy(e),
        particle_name(particleName), parent_ID(parentID)
        {}
```

g-2 Calorimeter Truth Hit

```
// gm2ringsim/calo/CaloArtRecord.hh

#include "CLHEP/Vector/ThreeVector.h"

namespace gm2ringsim {
    struct CaloArtRecord {
        int turn;
        int caloNum;
        int trackID;
        float x;        float y;        float z;
        float time;
        float px;        float py;        float pz;
        float energy;
        std::string particle_name;
        int parent_ID;

       CaloArtRecord() :   // This is better
          turn(), caloNum(), trackID(), x(), y(), z(), time(), px(), py(), pz(), energy(),
          particle_name("NOT_FILLED"), parent_ID()
          {}

// (rest left out)
```

Always explicitly initialize data in a default constructor

Data is immutable

A golden rule:

Data in the event, run or subrun cannot be changed (and the framework won't let you)!

You can add new data, but you can't change the data that's already there.

Why?

Allowing data to change in the event would add confusion, would be hard to track, would break provenance.

Science demands data integrity. Changing data breaks it

Review, reflect, contemplate, ask

What is data? How do we represent data in C++?

What does the framework do with data?

What is a data product? What is in a data product?

How can you find out what is in a data product?

Any questions?